Machine-Free Complexity
Implicit Complexity

Ugo Dal Lago

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Inria informatiques mathématiques

Caleidoscope Summer School, Paris, June 17-21, 2019

About This Course

1. **Introduction** to Implicit Computational Complexity.
   - Approximately one hour.

About This Course

1. **Introduction** to Implicit Computational Complexity.
    - Approximately one hour.
2. Implicit Complexity through the **Curry-Howard Correspondence**.
    - Approximately one hour.

About This Course

1. **Introduction** to Implicit Computational Complexity.
   - Approximately one hour.
2. Implicit Complexity through the **Curry-Howard Correspondence**.
   - Approximately one hour.
3. Boosting the **Intensional Expressive** Power of ICC Systems.
   - Approximately half an hour.

About This Course

1. **Introduction** to Implicit Computational Complexity.
   - Approximately one hour.
2. Implicit Complexity through the **Curry-Howard Correspondence**.
   - Approximately one hour.
3. Boosting the **Intensional Expressive** Power of ICC Systems.
   - Approximately half an hour.
4. **Challenges and Pespectives**: Probabilistic and Concurrent Computation.
   - Approximately half an hor.

About This Course

1. **Introduction** to Implicit Computational Complexity.
    - Approximately one hour.
2. Implicit Complexity through the **Curry-Howard Correspondence**.
    - Approximately one hour.
3. Boosting the **Intensional Expressive** Power of ICC Systems.
    - Approximately half an hour.
4. **Challenges and Pespectives**: Probabilistic and Concurrent Computation.
    - Approximately half an hor.

- **Website**: `http://www.cs.unibo.it/~dallago/CSCICC/`
- **Email**: `ugo.dallago@unibo.it`
    - Please contact me if you have any questions about the topics of this course.

Part I

A Short Introduction to Implicit Computational Complexity

Implicit Computational Complexity

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

Implicit Computational Complexity

- **Goal**
  - Machine-*free* characterizations of complexity classes.
  - No *explicit* reference to resource bounds.
  - P, PSPACE, L, NC,. . .
- **Why?**
  - Simple and elegant *presentations* of complexity classes.
  - *Formal methods* for complexity analysis of programs.

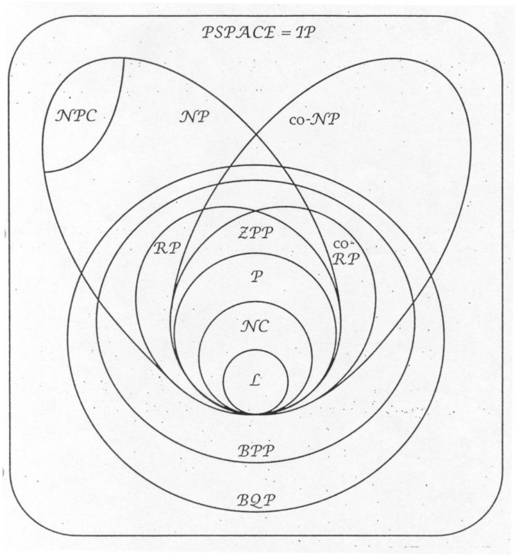
Implicit Computational Complexity

- **Goal**
  - Machine-*free* characterizations of complexity classes.
  - No *explicit* reference to resource bounds.
  - P, PSPACE, L, NC,...
- **Why?**
  - Simple and elegant *presentations* of complexity classes.
  - *Formal methods* for complexity analysis of programs.
- **How?**
  - Recursion Theory [BC92], [Leivant94], ...
  - Type Systems and λ-calculi [Hofmann97], ...
  - Proof Theory, via Cut-Elimination [Girard95], ...
  - ...

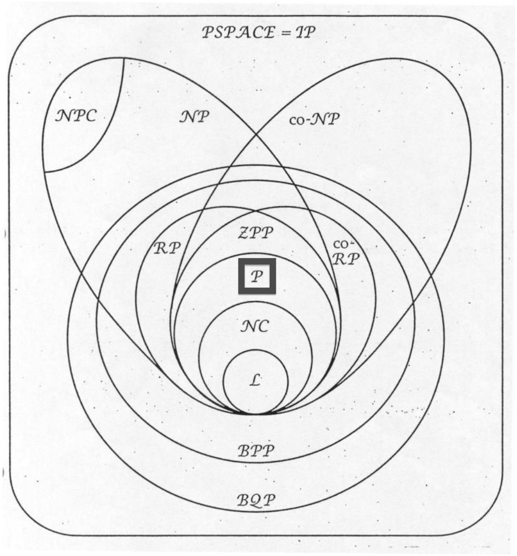
Complexity Classes

- Algorithms consume resources (**time**, **space**, **communication**, **energy**).
- How could one formalize the concept of being *efficient* with respect to a given resource?
- **A Measure**
  - One can assign to any algorithm P an upper bound on the amount of resources (of a certain kind) P needs when executed.
  - This can be either *precise* or *asymptotic*.
  - The more precise, the more *machine-dependent*.
- **A Predicate**
  - The amount of resources P needs when executed is bounded by a function in a "robust" class. Examples:
    - **Polynomial Functions**.
    - **Logarithmic Functions**.
    - **Elementary Functions**.
- **Complexity Class**: the set of those *functions* computed by *algorithms* satisfying the predicate.

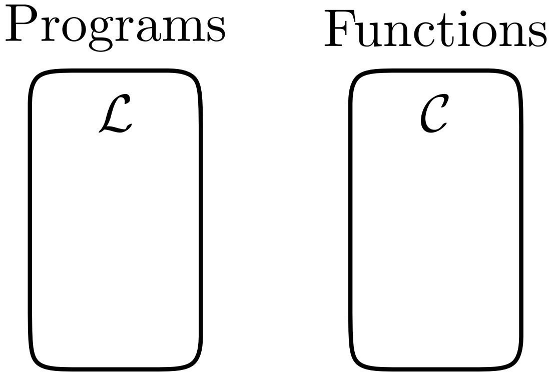
Complexity Classes



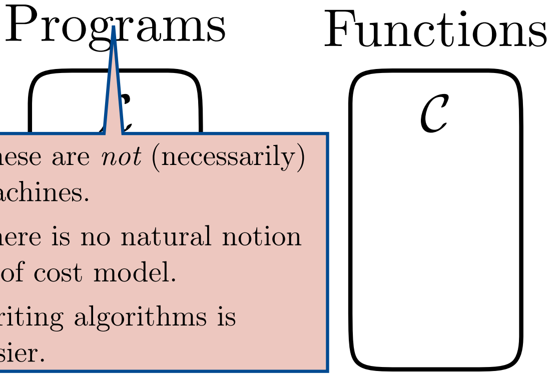
Complexity Classes



Characterizing Complexity Classes

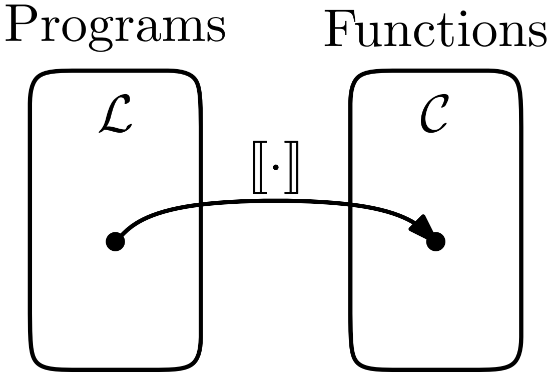


Characterizing Complexity Classes

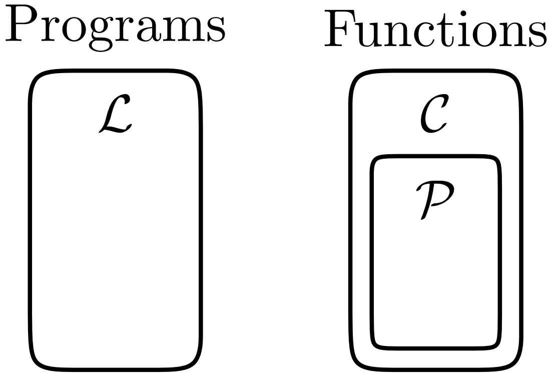


- These are *not* (necessarily) machines.
- There is no natural notion of of cost model.
- Writing algorithms is easier.

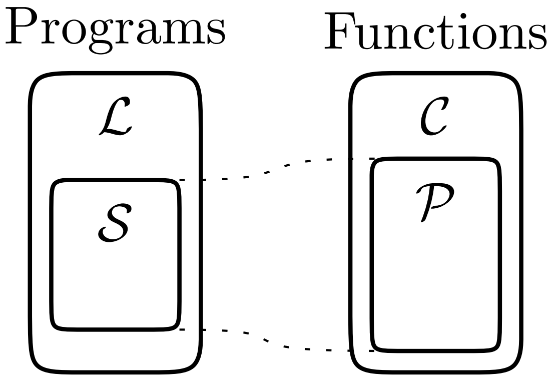
Characterizing Complexity Classes



Characterizing Complexity Classes



Characterizing Complexity Classes



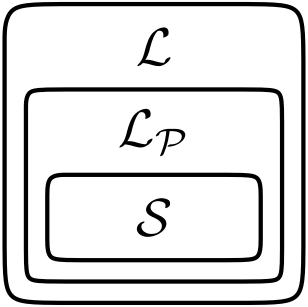
Proving $[\![\mathcal{S}]\!] = \mathcal{P}$

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

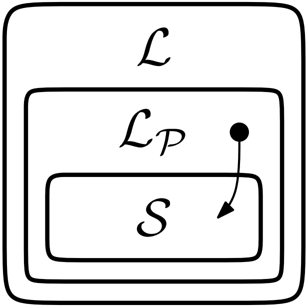
Proving $[\![\mathcal{S}]\!] = \mathcal{P}$

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

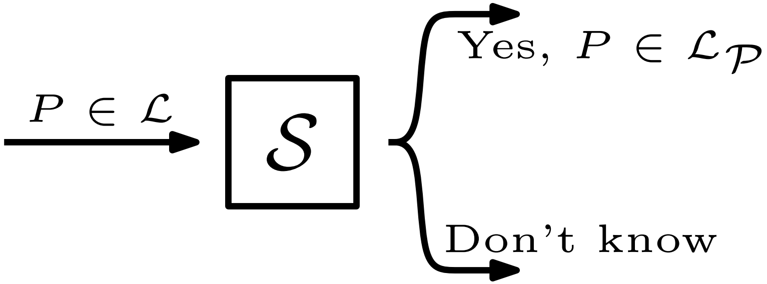
If Soundness is Proved Operationally...



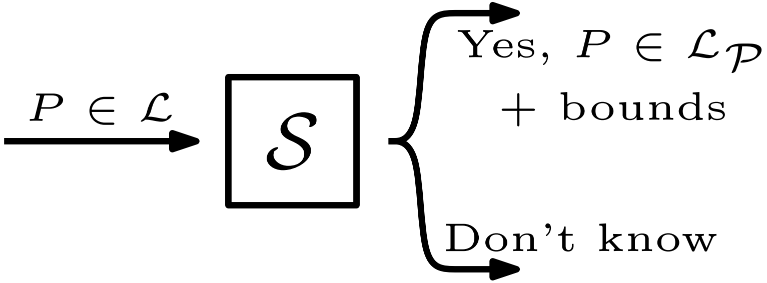
If Soundness is Proved Operationally...



ICC Systems as Static Analyzers



ICC Systems as Static Analyzers



Function Algebras

- Computation can be formalized by way of *machines*, like Turing machines, RAM, counter machines, etc.
  - *Defining* the amount of resources the machine consumes is **easy**.
  - *Writing* algorithms as machines is **difficult**.
  - Every such class of machines implicitly defines a class of **functions**.

Function Algebras

- Computation can be formalized by way of *machines*, like Turing machines, RAM, counter machines, etc.
    - *Defining* the amount of resources the machine consumes is **easy**.
    - *Writing* algorithms as machines is **difficult**.
    - Every such class of machines implicitly defines a class of **functions**.
- Another, different, way to capture computation is as follows:
    - Start from a set of *basic functions*...
    - ...and close it under some *operators*.

Function Algebras

- Computation can be formalized by way of *machines*, like Turing machines, RAM, counter machines, etc.
    - *Defining* the amount of resources the machine consumes is **easy**.
    - *Writing* algorithms as machines is **difficult**.
    - Every such class of machines implicitly defines a class of **functions**.
- Another, different, way to capture computation is as follows:
    - Start from a set of *basic functions*...
    - ...and close it under some *operators*.
- The set of **recursive functions** is the smallest set of functions which *contains* the basic functions and which is *closed* with respect to the operators.

Function Algebras

- Computation can be formalized by way of *machines*, like Turing machines, RAM, counter machines, etc.
  - *Defining* the amount of resources the machine consumes is **easy**.
  - *Writing* algorithms as machines is **difficult**.
  - Every such class of machines implicitly defines a class of **functions**.
- Another, different, way to capture computation is as follows:
  - Start from a set of *basic functions*...
  - ...and close it under some *operators*.
- The set of **recursive functions** is the smallest set of functions which *contains* the basic functions and which is *closed* with respect to the operators.
- Where is the **machine**? Where are the **programs**?
  - They are the *proofs of recursivity*, which are finitary.
  - As such, they support an induction principle.

Kleene's Function Algebra — Basic Functions

- **Zero**. The function $z : \mathbb{N} \to \mathbb{N}$ defined as follows: $z(n) = 0$ for every $n \in \mathbb{N}$.
- **Successor**. The function $s : \mathbb{N} \to \mathbb{N}$ defined as follows: $s(n) = n + 1$ for every $n \in \mathbb{N}$.
- **Projections**. For every positive $n \in \mathbb{N}$ and for whenever $1 \leq m \leq n$, the function $\Pi^n_m : \mathbb{N}^n \to \mathbb{N}$ is defined as follows: $\Pi^n_m(k_1, \ldots, k_n) = k_m$.

Kleeene's Function Algebra — Operations (I)

- **Composition**. Suppose that $n \in \mathbb{N}$ is positive, that $f : \mathbb{N}^n \to \mathbb{N}$ and that $g_m : \mathbb{N}^k \to \mathbb{N}$ for every $1 \leq m \leq n$. Then the *composition* of f and $g_1, \ldots, g_n$ is the function $h : \mathbb{N}^k \to \mathbb{N}$ defined as $h(\vec{i}) = f(g_1(\vec{i}), \ldots, g_n(\vec{i}))$.
- **Primitive Recursion**. Suppose that $n \in \mathbb{N}$ is positive, that $f : \mathbb{N}^n \to \mathbb{N}$ and that $g : \mathbb{N}^{n+2} \to \mathbb{N}$. Then the function $h : \mathbb{N}^{n+1} \to \mathbb{N}$ defined as follows

$$\begin{aligned} h(0, \vec{m}) &= f(\vec{m}); \\ h(k+1, \vec{m}) &= g(k, \vec{m}, h(k, \vec{m})); \end{aligned}$$

  is said to be defined by *primitive recursion* from f and g.

Kleene's Function Algebra — Operations (II)

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

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

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

Functional Programs — Preliminaries

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

Functional Programs — Preliminaries

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

Functional Programs

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

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

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

Functional Programs

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

$$
\begin{aligned}
P &::= R \mid R \, \mathbf{,} \, P \\
R &::= l \to t \\
l &::= \mathbf{f}_1 \, \mathbf{(} p_1^1 \mathbf{,} \, \ldots \, \mathbf{,} p_1^{\alpha(\mathbf{f}_1)} \mathbf{)} \mid \ldots \mid \mathbf{f}_n \, \mathbf{(} p_n^1 \mathbf{,} \, \ldots \, \mathbf{,} p_n^{\alpha(\mathbf{f}_n)} \mathbf{)}
\end{aligned}
$$

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

Functional Programs

- **Example**:

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

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

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

Functional Programs

- **Example**:

$$\begin{aligned}\mathtt{append}(\mathtt{e},x) &\to x\\ \mathtt{append}(\mathtt{0}(x),y) &\to \mathtt{0}(\mathtt{append}(x,y))\\ \mathtt{append}(\mathtt{1}(x),y) &\to \mathtt{1}(\mathtt{append}(x,y)).\end{aligned}$$

- **Example**:

$$\begin{aligned}\mathtt{reverse}(\mathtt{e}) &\to \mathtt{e}\\ \mathtt{reverse}(\mathtt{0}(x)) &\to \mathtt{1}(\mathtt{reverse}(x))\\ \mathtt{reverse}(\mathtt{1}(x)) &\to \mathtt{0}(\mathtt{reverse}(x)).\end{aligned}$$

- **Example**:

$$\begin{aligned}\mathtt{replicate}(\mathtt{e},x) &\to \mathtt{e}\\ \mathtt{replicate}(\mathtt{0}(x),y) &\to \mathtt{append}(\mathtt{replicate}(x,y),y)\\ \mathtt{replicate}(\mathtt{1}(x),y) &\to \mathtt{append}(\mathtt{replicate}(x,y),y).\end{aligned}$$

λ-Calculus

- **Terms**:
$$M ::= x \mid \lambda x.M \mid MM,$$
- The term obtained by *substituting* a term M for a variable x into another term N is denoted as $N\{M/x\}$.
- **Values**:
$$V ::= x \mid \lambda x.M.$$
- **Reduction**:
$$\frac{}{(\lambda x.M)V \rightarrow_v M\{V/x\}} \qquad \frac{M \rightarrow_v N}{ML \rightarrow_v NL} \qquad \frac{M \rightarrow_v N}{LM \rightarrow_v LN}$$
Here M ranges over terms, while V ranges over values.
- **Normal Forms**. Any term M such that there is not any N with $M \rightarrow_v N$. A term M *has* a normal form iff $M \rightarrow_v^* N$. Otherwise, we write $M \uparrow$.

λ-Calculus

- **Terms**:

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

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

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

- **Reduction**:

$$\frac{M \to_v N}{ML \to_v NL} \qquad \frac{M \to_v N}{LM \to_v LN}$$

ile V ranges over values.

V such that there is not any

has a normal form iff

$M \uparrow$.

- This is just one way of evaluating terms, the so called *weak call-by-value* evaluation.
- An alternative would be call-by-name, which however tends to be inefficient in many cases.

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

- **Scott's Numerals**.

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

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

Computability

- For the three introduced computational models, one can define the class of functions from $\mathbb{N}$ to $\mathbb{N}$ they compute.
- Unsurprisingly, they are all the same: the three models are all Turing-powerful.
- This means, in particular, that programs in the three formalisms can be, in general, **very inefficient**.
  - They can compute whatever (computable) function one can imagine, after all!

Computability

- For the three introduced computational models, one can define the class of functions from $\mathbb{N}$ to $\mathbb{N}$ they compute.
- Unsurprisingly, they are all the same: the three models are all Turing-powerful.
- This means, in particular, that programs in the three formalisms can be, in general, **very inefficient**.
    - They can compute whatever (computable) function one can imagine, after all!
- Is there a way to isolate the efficient recursive functions, functional programs, and λ-terms?
- First of all, we need to understand *what it means* for such things to be efficient.

Cost Models

- Let us focus our attention to *time* complexity.
- How do we *measure* the amount of time a computation takes in the three models we described?
    - **Functional Programs**: number of reduction steps?

Cost Models

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

Cost Models

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

Cost Models

- Let us focus our attention to *time* complexity.
- How do we *measure* the amount of time a computation takes in the three models we described?
  - **Functional Programs**: number of reduction steps?
  - **λ-Calculus**: number of reduction steps?
  - **Function Algebra**: ?
- **Invariance Thesis** [SvEB1980]: a cost model is *reasonable* iff the class of polytime computable functions is the same as the one defined with Turing programs.

Cost Models

- Let us focus our attention to *time* complexity.
- How do we *measure* the amount of time a computation takes in the three models we described?
  - **Functional Programs**: number of reduction steps?
  - **λ-Calculus**: number of reduction steps?
  - **Function Algebra**: ?
- **Invariance Thesis** [SvEB1980]: a cost model is *reasonable* iff the class of polytime computable functions is the same as the one defined with Turing programs.
- The most interesting cost models are, clearly, the unitary ones. They are not always invariant *by definition*, however.

Complexity Classes as Function Algebras?

- **Recursion on Notation**: since computation in unary notation is inefficient, we need to switch to binary notation.

Complexity Classes as Function Algebras?

- **Recursion on Notation**: since computation in unary notation is inefficient, we need to switch to binary notation.
- **What Should we Keep in the Algebra**?
  - Basic functions are innoquous.
  - Polytime functions are closed by composition.
  - Minimization introduces partiality, and is not needed.
  - Primitive Recursion?

Complexity Classes as Function Algebras?

- **Recursion on Notation**: since computation in unary notation is inefficient, we need to switch to binary notation.
- **What Should we Keep in the Algebra**?
  - Basic functions are innoquous.
  - Polytime functions are closed by composition.
  - Minimization introduces partiality, and is not needed.
  - Primitive Recursion?
- *Certain* uses of primitive recursion are dangerous, but if we *do not* have any form of recursion, we capture much less than polytime functions.

Safe Recursion [BellantoniCook93]

- **Safe Functions**: pairs in the form (f, n), where $f : \mathbb{B}^m \to \mathbb{B}$ and $0 \leq n \leq m$.
- The number n identifies the number of *normal arguments* between those of f: they are the first n, while the other $m - n$ are the *safe arguments*.
- Following [BellantoniCook93], we use semicolons to separate normal and safe arguments: if (f, n) is a safe function, we write $f(\vec{W}; \vec{V})$ to emphasize that the n words in $\vec{W}$ are the normal arguments, while the ones in $\vec{V}$ are the safe arguments.

Basic Safe Functions

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

Safe Composition

$$\boxed{\begin{array}{c} (f : \mathbb{B}^n \to \mathbb{B}, m) \\ (g_j : \mathbb{B}^k \to \mathbb{B}, k) \text{ for every } 1 \leq j \leq m \\ (h_j : \mathbb{B}^{k+i} \to \mathbb{B}, k) \text{ for every } m+1 \leq j \leq n \end{array}}$$

Safe Composition

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

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

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

$$\Downarrow$$

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

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

(Simultaneous) Safe Recursion

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

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

(Simultaneous) Safe Recursion

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

$$\Downarrow$$

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

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

Classes of Functions

- BCS is the smallest class of safe functions which includes the basic safe functions above and which is closed by safe composition and safe recursion.
- BC is the set of those functions $f : \mathbb{B} \to \mathbb{B}$ such that $(f, n) \in$ BCS for some $n \in \{0, 1\}$.

Lemma (Max-Poly Lemma)

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

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

Lemma (Max-Poly Lemma)

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

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

Proof.

- An induction on the structure of the proof a function being in BCS.
- The restriction to *safe* recursion is *essential.*

$\square$

Theorem (Polytime Soundness)

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

Theorem (Polytime Soundness)

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

Proof.

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

□

Lemma (Polynomials)

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

Lemma (Polynomials)

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

Theorem (Polytime Completeness)

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

Lemma (Polynomials)

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

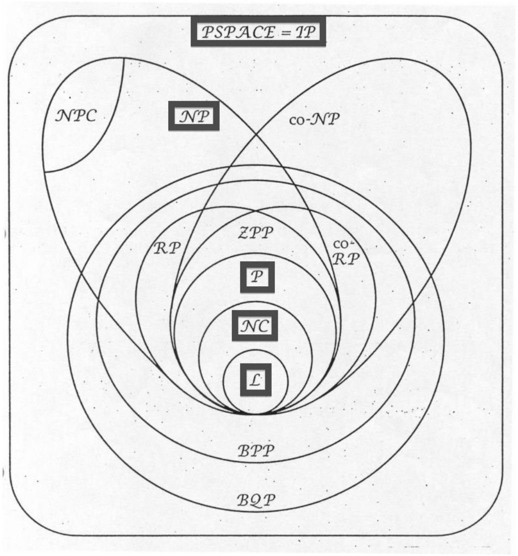
Theorem (Polytime Completeness)

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

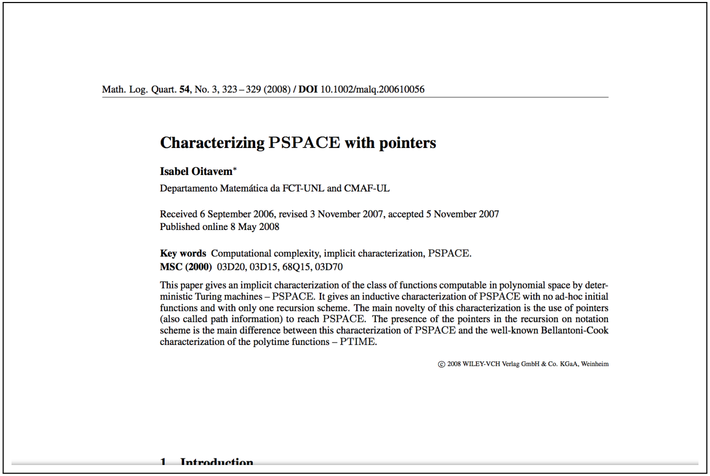
Theorem (BellantoniCook1992)

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

Further Work



Further Work



Math. Log. Quart. **54**, No. 3, 323 – 329 (2008) / **DOI** 10.1002/malq.200610056

Characterizing PSPACE with pointers

Isabel Oitavem*

Departamento Matemática da FCT-UNL and CMAF-UL

Received 6 September 2006, revised 3 November 2007, accepted 5 November 2007
Published online 8 May 2008

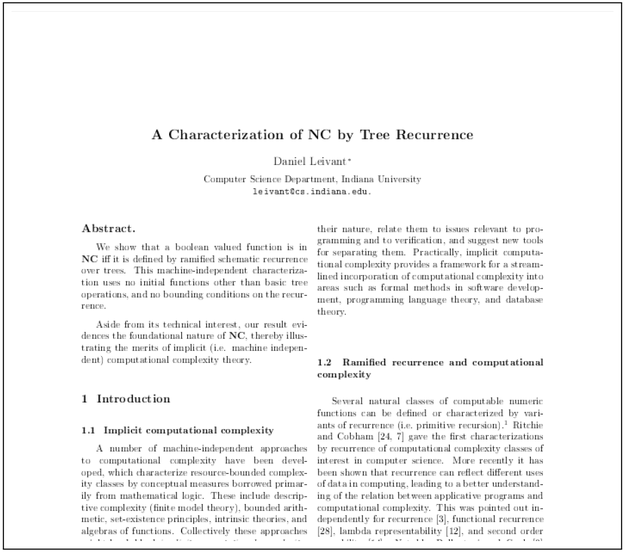
Key words Computational complexity, implicit characterization, PSPACE.
MSC (2000) 03D20, 03D15, 68Q15, 03D70

This paper gives an implicit characterization of the class of functions computable in polynomial space by deterministic Turing machines – PSPACE. It gives an inductive characterization of PSPACE with no ad-hoc initial functions and with only one recursion scheme. The main novelty of this characterization is the use of pointers (also called path information) to reach PSPACE. The presence of the pointers in the recursion on notation scheme is the main difference between this characterization of PSPACE and the well-known Bellantoni-Cook characterization of the polytime functions – PTIME.

1 Introduction

PSPACE

Further Work



A Characterization of NC by Tree Recurrence

Daniel Leivant*

Computer Science Department, Indiana University

`leivant@cs.indiana.edu.`

Abstract.

We show that a boolean valued function is in **NC** iff it is defined by ramified schematic recurrence over trees. This machine-independent characterization uses no initial functions other than basic tree operations, and no bounding conditions on the recurrence.

Aside from its technical interest, our result evidences the foundational nature of **NC**, thereby illustrating the merits of implicit (i.e. machine independent) computational complexity theory.

1 Introduction

1.1 Implicit computational complexity

A number of machine-independent approaches to computational complexity have been developed, which characterize resource-bounded complexity classes by conceptual measures borrowed primarily from mathematical logic. These include descriptive complexity (finite model theory), bounded arithmetic, set-existence principles, intrinsic theories, and algebras of functions. Collectively these approaches

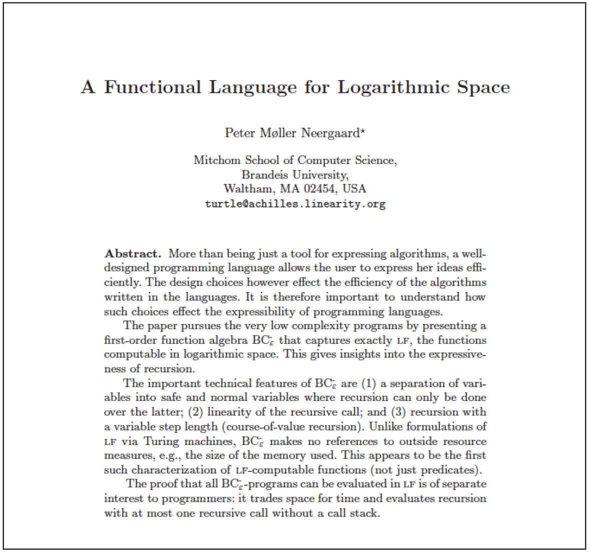
their nature, relate them to issues relevant to programming and to verification, and suggest new tools for separating them. Practically, implicit computational complexity provides a framework for a streamlined incorporation of computational complexity into areas such as formal methods in software development, programming language theory, and database theory.

1.2 Ramified recurrence and computational complexity

Several natural classes of computable numeric functions can be defined or characterized by variants of recurrence (i.e. primitive recursion).[1] Ritchie and Cobham [24, 7] gave the first characterizations by recurrence of computational complexity classes of interest in computer science. More recently it has been shown that recurrence can reflect different uses of data in computing, leading to a better understanding of the relation between applicative programs and computational complexity. This was pointed out independently for recurrence [3], functional recurrence [28], lambda representability [12], and second order

NC

Further Work



A Functional Language for Logarithmic Space

Peter Møller Neergaard*

Mitchom School of Computer Science,
Brandeis University,
Waltham, MA 02454, USA
`turtle@achilles.linearity.org`

Abstract. More than being just a tool for expressing algorithms, a well-designed programming language allows the user to express her ideas efficiently. The design choices however effect the efficiency of the algorithms written in the languages. It is therefore important to understand how such choices effect the expressibility of programming languages.

The paper pursues the very low complexity programs by presenting a first-order function algebra BC_ε^- that captures exactly LF, the functions computable in logarithmic space. This gives insights into the expressiveness of recursion.

The important technical features of BC_ε^- are (1) a separation of variables into safe and normal variables where recursion can only be done over the latter; (2) linearity of the recursive call; and (3) recursion with a variable step length (course-of-value recursion). Unlike formulations of LF via Turing machines, BC_ε^- makes no references to outside resource measures, e.g., the size of the memory used. This appears to be the first such characterization of LF-computable functions (not just predicates).

The proof that all BC_ε^--programs can be evaluated in LF is of separate interest to programmers: it trades space for time and evaluates recursion with at most one recursive call without a call stack.

LOGSPACE

Which Cost Model?

- Is it sensible to take the number of reduction steps as a measure of the execution time of a functional program P?

Which Cost Model?

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

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

Which Cost Model?

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

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

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

Which Cost Model?

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

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

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

Theorem (DLMartini2009)

The unitary cost model is invariant, both in functional program and in the λ-calculus.

The Interpretation Method

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

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

  which is strictly increasing in any of its argument w.r.t. $\leq$.
- Given an assignment $\mathcal{A}$, one can generalize it to a map on closed and open terms.
- **Interpretation** for a functional program P: an assignment such that for every rule $l \to t$, it holds that

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

Theorem (Lankford1979)

A functional program P is terminating iff there is one interpretation for it.

Polynomial Interpretations

- What if we choose ℕ as the underlying domain, and polynomials on the natural numbers as the functions intepreting them?

Polynomial Interpretations

- What if we choose $\mathbb{N}$ as the underlying domain, and polynomials on the natural numbers as the functions intepreting them?
- Do we get a characterization of polynomial time computable functions?

Polynomial Interpretations

- What if we choose $\mathbb{N}$ as the underlying domain, and polynomials on the natural numbers as the functions intepreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that $\mathbf{f}$ is a unary function symbol, and that t is a closed term in, say, $\mathcal{C}(\mathcal{S}_{\mathbb{B}})$.

Polynomial Interpretations

- What if we choose $\mathbb{N}$ as the underlying domain, and polynomials on the natural numbers as the functions intepreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that $\mathbf{f}$ is a unary function symbol, and that t is a closed term in, say, $\mathcal{C}(\mathcal{S}_{\mathbb{B}})$.
  - If $\mathbf{f}(t) \rightarrow^n s$, then $n \leq [\mathbf{f}(t)]_{\mathcal{A}} = [\mathbf{f}]_{\mathcal{A}}([t]_{\mathcal{A}})$

Polynomial Interpretations

- What if we choose $\mathbb{N}$ as the underlying domain, and polynomials on the natural numbers as the functions intepreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that $\mathbf{f}$ is a unary function symbol, and that t is a closed term in, say, $\mathcal{C}(\mathcal{S}_{\mathbb{B}})$.
  - If $\mathbf{f}(t) \rightarrow^{n} s$, then $n \leq [\mathbf{f}(t)]_{\mathcal{A}} = [\mathbf{f}]_{\mathcal{A}}([t]_{\mathcal{A}})$
  - But $[t]_{\mathcal{A}}$ can be *much* bigger than $|t|$.

Polynomial Interpretations

- What if we choose $\mathbb{N}$ as the underlying domain, and polynomials on the natural numbers as the functions intepreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that $\mathbf{f}$ is a unary function symbol, and that t is a closed term in, say, $\mathcal{C}(\mathcal{S}_{\mathbb{B}})$.
  - If $\mathbf{f}(t) \rightarrow^n s$, then $n \leq [\mathbf{f}(t)]_{\mathcal{A}} = [\mathbf{f}]_{\mathcal{A}}([t]_{\mathcal{A}})$
  - But $[t]_{\mathcal{A}}$ can be *much* bigger than $|t|$.
  - Everything depends on the way you interpret data!

Polynomial Interpretations

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

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

Theorem (BCMT2001)

Additive polynomial intepretations characterize polynomial time computable functions.

Other Techniques

- **Path Orders**
  - Originally introduced for termination.
  - PPO, MPO, LPO, ...
  - Later refined to guarantee polytime complexity.
    - LMPO [Marion2003].
    - POP* [AM2008].
- **Quasi-Interpretations**
  - Obtained by combining interpretations and path orders [BMM2011].
  - The value of the lhs can be *equal* to the rhs.
  - Intensionally very powerful.
  - Various classes can be characterized (in particular, FP and FPSPACE).
  - Not sound for the unitary cost model.

Part II

Implicit Complexity and the Curry-Howard Correspondence

Definition

- A *direct relationship* between:
  - **Programs**, seen as computational objects.
  - **Proofs**, seen as logical objects.
- This extends to various key concepts in logic and programming:

Logic	**Programming**
Formulas	*Types*
Implication	*Function Type*
Introduction Rule	*Constructor*
Elimination Rule	*Destructor*
Normalization	*Computation*

Examples — Hilbert-Style Proofs

- **Combinatory Logic**

$$M ::= x \mid \mathbf{S} \mid \mathbf{K} \mid MM$$

- **Typed CL vs. Hilbert-style Proofs**

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}$$

$$\frac{}{\Gamma \vdash \alpha \to (\beta \to \alpha)} \qquad \frac{}{\Gamma \vdash \mathbf{K} : \alpha \to (\beta \to \alpha)}$$

$$\frac{}{\Gamma \vdash \begin{array}{l}(\alpha \to (\beta \to \gamma)) \to \\ (\alpha \to \beta) \to (\alpha \to \gamma)\end{array}} \qquad \frac{}{\Gamma \vdash \mathbf{S} : \begin{array}{l}(\alpha \to (\beta \to \gamma)) \to \\ (\alpha \to \beta) \to (\alpha \to \gamma)\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \alpha \to \beta \\ \Gamma \vdash \alpha\end{array}}{\Gamma \vdash \beta} \qquad \frac{\begin{array}{c}\Gamma \vdash M : \alpha \to \beta \\ \Gamma \vdash N : \alpha\end{array}}{\Gamma \vdash MN : \beta}$$

Examples — Natural Deduction

- **λ-Calculus**

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

- **Typed λ-Calculus vs. Natural Deduction**

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}$$

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta} \qquad \frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x.M : \alpha \to \beta}$$

$$\frac{\begin{array}{c}\Gamma \vdash \alpha \to \beta \\ \Gamma \vdash \alpha\end{array}}{\Gamma \vdash \beta} \qquad \frac{\begin{array}{c}\Gamma \vdash M : \alpha \to \beta \\ \Gamma \vdash N : \alpha\end{array}}{\Gamma \vdash MN : \beta}$$

Examples — Extensions

- **Intuitionistic Second-Order Logic**
  - One can endow the underlying *logic* with conjunctions, disjunctions, and second order quantification.
  - On the side of *programs*, one could proceed by adding pairs, projections, etc.
  - This way, the correspondence continues to hold.
  - The obtained language, called $\mathbb{F}$, enjoys strong normalization.

Examples — Extensions

- **Intuitionistic Second-Order Logic**
  - One can endow the underlying *logic* with conjunctions, disjunctions, and second order quantification.
  - On the side of *programs*, one could proceed by adding pairs, projections, etc.
  - This way, the correspondence continues to hold.
  - The obtained language, called $\mathbb{F}$, enjoys strong normalization.
- **Heyting Arithmetic**
  - One can endow (intuitionistic) first-order logic with the Peano's axioms for the natural numbers, and an induction principle.
  - The obtained formal system can be, following Gödel, be *realized* by terms in an extension of the simply-typed λ-calculus:

$$\frac{}{\Gamma \vdash \mathbf{Z} : nat} \qquad \frac{}{\Gamma \vdash \mathbf{S} : nat \to nat}$$

$$\frac{}{\Gamma \vdash \mathbf{R} : nat \to (nat \to \alpha \to \alpha) \to \alpha}$$

  - Again the obtained language, called $\mathbb{T}$, enjoys strong normalization.

Complexity?

- **Simply-Typed λ-Calculus**
  - Data are represented via their Church-encoding, e.g.

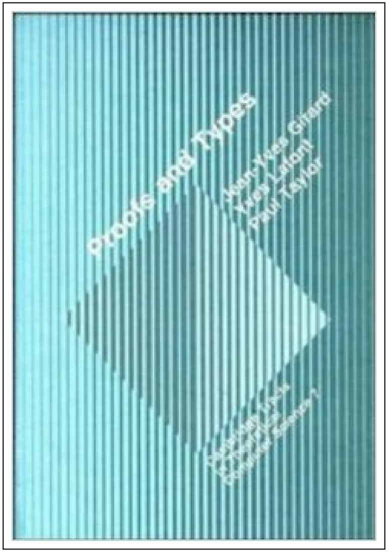
$$nat_\alpha = (\alpha \to \alpha) \to \alpha \to \alpha$$
$$\ulcorner n \urcorner = \lambda x.\lambda y.\ \underbrace{x \ldots x}_{n \text{ times}}\ y$$

  - On the one hand, normalization is known to have at least elementary complexity [Statman1977,FLOD1983]
  - On the other hand, equality cannot be represented [Statman1982]
- **System $\mathbb{F}$**
  - We can go polymorphic:

$$nat = \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$$

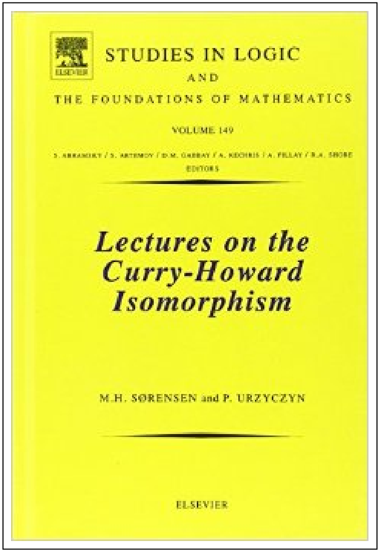
  - The representable functions are the ones which are provably total in second-order arithmetic.
- **System $\mathbb{T}$**
  - The fact the language realizes HA implies that the representable functions are the provably total functions of PA.

Further Readings



SURVEY

Further Readings



STUDIES IN LOGIC
AND
THE FOUNDATIONS OF MATHEMATICS

VOLUME 149

S. ABRAMSKY / S. ARTEMOV / D.M. GABBAY / A. KECHRIS / A. PILLAY / R.A. SHORE
EDITORS

Lectures on the Curry-Howard Isomorphism

M.H. SØRENSEN and P. URZYCZYN

ELSEVIER

REFERENCE

λ-Calculus and Complexity

- Type Systems induced by "mainstream" logics are simply too powerful to be easily turned into ICC systems.
- How should we proceed to, e.g. isolate a class of λ-terms computing polytime functions?

λ-Calculus and Complexity

- Type Systems induced by “mainstream” logics are simply too powerful to be easily turned into ICC systems.
- How should we proceed to, e.g. isolate a class of λ-terms computing polytime functions?
- **Type Systems** [Hofmann1997, BNS2000, Hofmann1999].
  - Start from something similar to $\mathbb{T}$.
  - Then, impose some constraints on recursion, akin to those from [BellantoniCook1993].

λ-Calculus and Complexity

- Type Systems induced by "mainstream" logics are simply too powerful to be easily turned into ICC systems.
- How should we proceed to, e.g. isolate a class of λ-terms computing polytime functions?
- **Type Systems** [Hofmann1997, BNS2000, Hofmann1999].
  - Start from something similar to $\mathbb{T}$.
  - Then, impose some constraints on recursion, akin to those from [BellantoniCook1993].
- **Linearity Constraints** [Girard1997, Lafont2004].
  - Key observation: copying is the operation making evaluation of λ-expressions problematic from a complexity point of view.
  - Let us define some constraints on duplication, then!

Safe Recursion and Type Systems

- Could we easily turn safe recursion into a type system guaranteeing polynomial time computability in, e.g., System $\mathbb{T}$?
- **Naive Idea**: use two function spaces.
  - The space $\square A \to B$ of safe functions
  - The space $\blacksquare A \to B$ of normal functions.

  And apply the same restrictions as in safe recursion, getting SR

Safe Recursion and Type Systems

- Could we easily turn safe recursion into a type system guaranteeing polynomial time computability in, e.g., System $\mathbb{T}$?
- **Naive Idea**: use two function spaces.
  - The space $\Box A \to B$ of safe functions
  - The sp

  And apply ng SR

- Typing rules are those of $\mathbb{T}$.
- Iteration can be typed as follows

$$\blacksquare nat \to (\Box A \to A) \to \Box A \to A$$

Safe Recursion and Type Systems

- Could we easily turn safe recursion into a type system guaranteeing polynomial time computability in, e.g., System $\mathbb{T}$?
- **Naive Idea**: use two function spaces.
  - The space $\square A \to B$ of safe functions
  - The space $\blacksquare A \to B$ of normal functions.

  And apply the same restrictions as in safe recursion, getting SR
- Unfortunately, this does not work!

$$\begin{aligned} HOexp(0) &= \lambda x.\mathbf{S}\ x; \\ HOexp(n+1) &= \lambda x.HOexp(n)(HOexp(n)(x)). \end{aligned}$$

Safe Recursion and Type Systems

- Could we easily turn safe recursion into a type system guaranteeing polynomial time computability in, e.g., System $\mathbb{T}$?
- **Naive Idea**: use two function spaces.
  - The space $\Box A \rightarrow B$ of safe functions
  - The space $\blacksquare A \rightarrow B$ of normal functions.

  And apply the same restrictions as in safe recursion, getting SR
- Unfortunately, this does not work!

$$
\begin{aligned}
HOexp(0) &= \lambda x.\mathbf{S}\ x; \\
HOexp(n+1) &= \lambda x.HOexp(n)(HOexp(n)(x)).
\end{aligned}
$$

- **Key Insight**: higher-order variables, essentially, should be very hard to copy.
  - Well, at least those involved in recursive definitions.

Theorem

SLR *precisely captures* **FP**.

Safe Recursion and Type Systems

- Could we easily turn safe recursion into a type system guaranteeing polynomial time computability in, e.g., System $\mathbb{T}$?
- **Naive Idea**: use two function spaces.
  - The space $\square A \to B$ of safe functions
  - The space $\blacksquare A \to B$ of normal functions.

  And apply the same restrictions as in safe recursion, getting SR

- The type of iteration becomes

$$\blacksquare nat \to (\square A \multimap A) \to \square A \to A$$

where $A \multimap B$ is the type of *affine functions* from A to B.

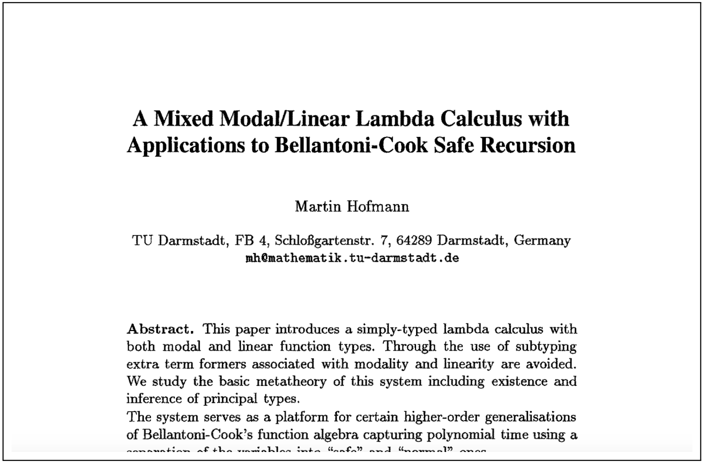
$(HOexp(n)(x))$.

essentially, should be very ha to copy.

  - Well, at least those involved in recursive definitions.

Theorem

SLR *precisely captures* **FP**.


A Mixed Modal/Linear Lambda Calculus with Applications to Bellantoni-Cook Safe Recursion

Martin Hofmann

TU Darmstadt, FB 4, Schloßgartenstr. 7, 64289 Darmstadt, Germany
mh@mathematik.tu-darmstadt.de

Abstract. This paper introduces a simply-typed lambda calculus with both modal and linear function types. Through the use of subtyping extra term formers associated with modality and linearity are avoided. We study the basic metatheory of this system including existence and inference of principal types.
The system serves as a platform for certain higher-order generalisations of Bellantoni-Cook's function algebra capturing polynomial time using a

SLR

Capturing More Algorithms

- Safe recursion is sound and complete for polynomially computable functions.

Capturing More Algorithms

- Safe recursion is sound and complete for polynomially computable functions.
- Completeness, however, only holds in a purely **extensional** sense.
  - Not all polynomial time *algorithms* can be captured.

Capturing More Algorithms

- Safe recursion is sound and complete for polynomially computable functions.
- Completeness, however, only holds in a purely **extensional** sense.
  - Not all polynomial time *algorithms* can be captured.
- (Linear) safe recursion, as an example, rules out *all* nested recursive definitions, and then many useful programs
  - Example: `InsertionSort`.

Capturing More Algorithms

- Safe recursion is sound and complete for polynomially computable functions.
- Completeness, however, only holds in a purely **extensional** sense.
  - Not all polynomial time *algorithms* can be captured.
- (Linear) safe recursion, as an example, rules out *all* nested recursive definitions, and then many useful programs
  - Example: `InsertionSort`.
- **Key Insight**: one should also enforce functions to be *non-size increasing*:

$$\frac{\vdash M : \Box A \multimap A \quad \vdash N : A}{\vdash \mathtt{rec}(M, N) : \blacksquare NAT \to A} \qquad \frac{}{\vdash \mathtt{succ} : \Box NAT \to NAT}$$

Capturing More Algorithms

- Safe recursion is sound and complete for polynomially computable functions.
- Completeness, however, only holds in a purely **extensional** sense.
  - Not all polynomial time *algorithms* can be captured.
- (Linear) safe recursion, as an example, rules out *all* nested recursive definitions, and then many useful programs
  - Example: `InsertionSort`.
- **Key Insight**: one should also enforce functions to be *non-size increasing*:

$$\frac{\vdash M : \square A \multimap A \quad \vdash N : A}{\vdash \mathtt{rec}(M,N) : \blacksquare NAT \to A} \qquad \frac{}{\vdash \mathtt{succ} : \square NAT \to NAT}$$

$$\Downarrow$$

$$\frac{\vdash M : \diamond \multimap A \multimap A \quad \vdash N : A}{\vdash \mathtt{rec}(M,N) : NAT \multimap A} \qquad \frac{}{\vdash \mathtt{succ} : \diamond \multimap NAT \multimap NAT}$$

Capturing More Algorithms

- Safe recursion is sound and complete for polynomially computable functions.
- Completeness, however, only holds in a purely **extensional** sense.
  - Not all polynomial time *algorithms* can be captured.
- (Linear) safe recursion, as an example, rules out *all* nested recursive definitions, and then many useful programs
  - Example: `InsertionSort`.
- **Key Insight**: one should also enforce functions to be *non-size increasing*:

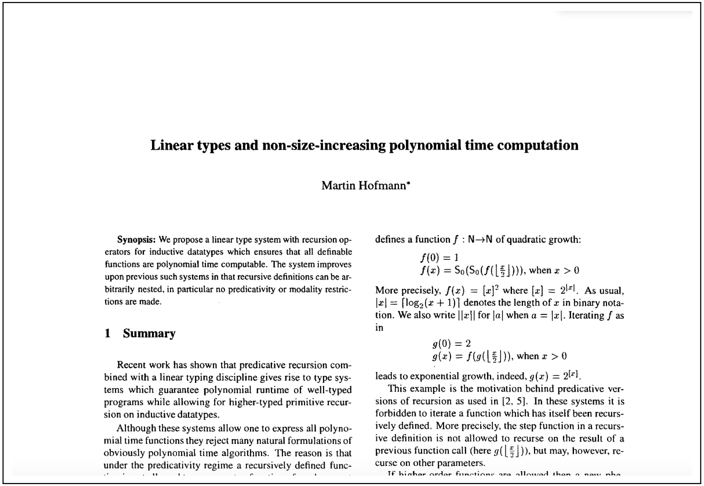
$$\frac{\vdash M : \square A \multimap A \quad \vdash N : A}{\vdash \mathtt{rec}(M,N) : \blacksquare NAT \to A} \qquad \frac{}{\vdash \mathtt{succ} : \square NAT \to NAT}$$

$$\Downarrow$$

$$\frac{\vdash M : \diamond \multimap A \multimap A \quad \vdash N : A}{\vdash \mathtt{rec}(M,N) : NAT \multimap A} \qquad \frac{}{\vdash \mathtt{succ} : \diamond \multimap NAT \multimap NAT}$$

Theorem

The obtained system, called LFPL*, is sound and complete for* **P**.



Linear types and non-size-increasing polynomial time computation

Martin Hofmann*

Synopsis: We propose a linear type system with recursion operators for inductive datatypes which ensures that all definable functions are polynomial time computable. The system improves upon previous such systems in that recursive definitions can be arbitrarily nested, in particular no predicativity or modality restrictions are made.

1 Summary

Recent work has shown that predicative recursion combined with a linear typing discipline gives rise to type systems which guarantee polynomial runtime of well-typed programs while allowing for higher-typed primitive recursion on inductive datatypes.

Although these systems allow one to express all polynomial time functions they reject many natural formulations of obviously polynomial time algorithms. The reason is that under the predicativity regime a recursively defined func-

defines a function $f : \mathsf{N} \to \mathsf{N}$ of quadratic growth:

$$\begin{aligned} f(0) &= 1 \\ f(x) &= \mathsf{S}_0(\mathsf{S}_0(f(\lfloor \tfrac{x}{2} \rfloor))), \text{ when } x > 0 \end{aligned}$$

More precisely, $f(x) = [x]^2$ where $[x] = 2^{|x|}$. As usual, $|x| = \lceil \log_2(x+1) \rceil$ denotes the length of x in binary notation. We also write $||x||$ for $|a|$ when $a = |x|$. Iterating f as in

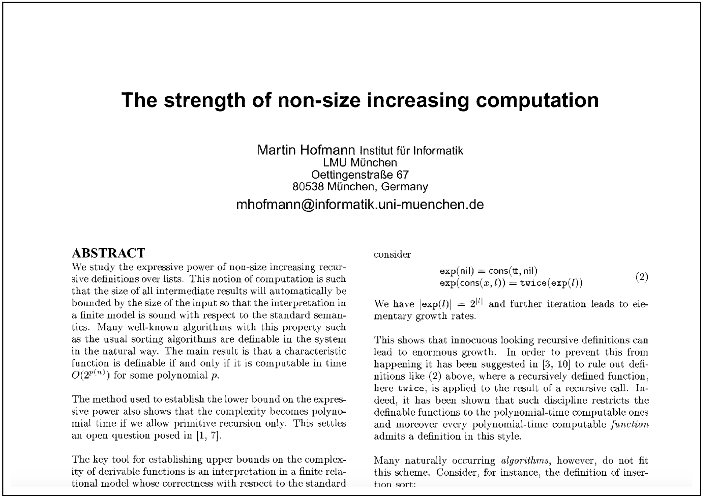
$$\begin{aligned} g(0) &= 2 \\ g(x) &= f(g(\lfloor \tfrac{x}{2} \rfloor)), \text{ when } x > 0 \end{aligned}$$

leads to exponential growth, indeed, $g(x) = 2^{[x]}$.

This example is the motivation behind predicative versions of recursion as used in [2, 5]. In these systems it is forbidden to iterate a function which has itself been recursively defined. More precisely, the step function in a recursive definition is not allowed to recurse on the result of a previous function call (here $g(\lfloor \frac{x}{2} \rfloor)$), but may, however, recurse on other parameters.

If higher order functions are allowed then a new phe

LFPL



The strength of non-size increasing computation

Martin Hofmann Institut für Informatik
LMU München
Oettingenstraße 67
80538 München, Germany
mhofmann@informatik.uni-muenchen.de

ABSTRACT

We study the expressive power of non-size increasing recursive definitions over lists. This notion of computation is such that the size of all intermediate results will automatically be bounded by the size of the input so that the interpretation in a finite model is sound with respect to the standard semantics. Many well-known algorithms with this property such as the usual sorting algorithms are definable in the system in the natural way. The main result is that a characteristic function is definable if and only if it is computable in time $O(2^{p(n)})$ for some polynomial p.

The method used to establish the lower bound on the expressive power also shows that the complexity becomes polynomial time if we allow primitive recursion only. This settles an open question posed in [1, 7].

The key tool for establishing upper bounds on the complexity of derivable functions is an interpretation in a finite relational model whose correctness with respect to the standard

consider

$$\begin{aligned}\mathtt{exp}(\mathsf{nil}) &= \mathsf{cons}(\mathsf{tt}, \mathsf{nil})\\ \mathtt{exp}(\mathsf{cons}(x,l)) &= \mathtt{twice}(\mathtt{exp}(l))\end{aligned} \qquad (2)$$

We have $|\mathtt{exp}(l)| = 2^{|l|}$ and further iteration leads to elementary growth rates.

This shows that innocuous looking recursive definitions can lead to enormous growth. In order to prevent this from happening it has been suggested in [3, 10] to rule out definitions like (2) above, where a recursively defined function, here twice, is applied to the result of a recursive call. Indeed, it has been shown that such discipline restricts the definable functions to the polynomial-time computable ones and moreover every polynomial-time computable *function* admits a definition in this style.

Many naturally occurring *algorithms*, however, do not fit this scheme. Consider, for instance, the definition of insertion sort:

LFPL

From Lambda Calculus to Soft Lambda Calculus

- Lambda calculus Λ:

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

  with no structural constraints.

From Lambda Calculus to Soft Lambda Calculus

- Lambda calculus Λ:

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

  with no structural constraints.
- Linear Lambda Calculus $\Lambda_!$

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

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

$$(\lambda x.M)N \to M\{N/x\} \qquad\qquad (\lambda !x.M)!N \to M\{N/x\}$$

From Lambda Calculus to Soft Lambda Calculus

- Lambda calculus Λ:

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

with no structural constraints.

- Linear Lambda Calculus $\Lambda_!$

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

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

$$(\lambda x.M)N \to M\{N/x\} \qquad\qquad (\lambda !x.M)!N \to M\{N/x\}$$

- Soft Lambda Calculus Λ_S

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

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

From Lambda Calculus to Soft Lambda Calculus

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

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

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

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

  - Some results:
    - Polytime **soundness**;
    - Polytime **completeness**.

Linear Logic

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

$$\begin{array}{rlcl}
!A & \multimap !A \otimes !A & \qquad & \lambda !x.\langle x, x\rangle \\
!A \otimes !B & \multimap !(A \otimes B) & & \lambda !x.\lambda !y.!\langle x, y\rangle \\
!A & \multimap !!A & & \lambda !x.!!x \\
!A & \multimap A & & \lambda !x.x
\end{array}$$

Linear Logic

Subsystems...

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

Linear Logic

Subsystems...

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

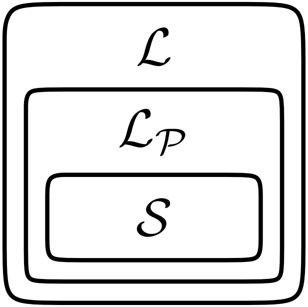
...and their expressive power

ELL	Elementary Functions
LLL	Polytime Functions
SLL	Polytime Functions

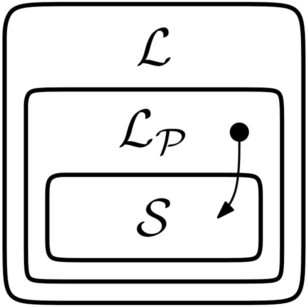
Part III

Boosting the Expressive Power of ICC Systems

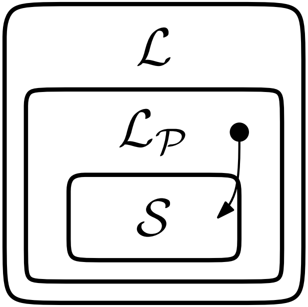
ICC: Intensional Expressive Power



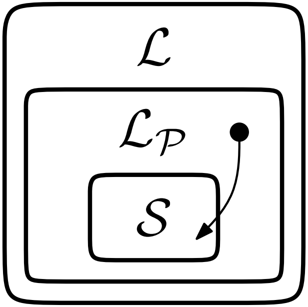
ICC: Intensional Expressive Power



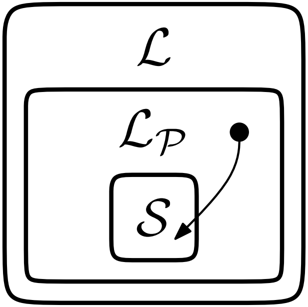
ICC: Intensional Expressive Power



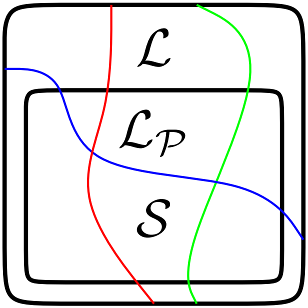
ICC: Intensional Expressive Power



ICC: Intensional Expressive Power



ICC: Intensional Expressive Power



Safe Recursion [BC93]

Light Linear Logic [Girard97]

⋮

How Many Terms Could we Catch?

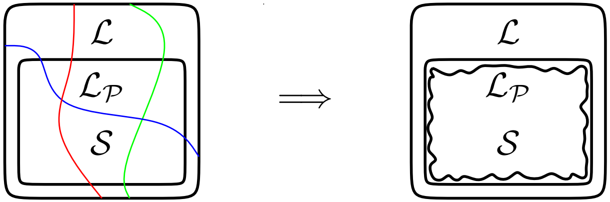
- Very Few, Actually!

How Many Terms Could we Catch?

- Very Few, Actually!
- In BC one cannot capture any of the *interesting* algorithms computing the sorting functions, like `QuickSort`, `MergeSort`, `InsertionSort`.

How Many Terms Could we Catch?

- Very Few, Actually!
- In BC one cannot capture any of the *interesting* algorithms computing the sorting functions, like `QuickSort`, `MergeSort`, `InsertionSort`.
- **Idea**:



Bounded Linear Logic.

- It is a refinement on (intuitionistic) linear logic:

$$
\begin{aligned}
!\ A &\multimap 1 \\
!\quad A &\multimap A \\
!\quad A &\multimap !\ A \otimes !\ A \\
!\quad A &\multimap !\ !\ A
\end{aligned}
$$

Bounded Linear Logic.

- It is a refinement on (intuitionistic) linear logic:

$$
\begin{aligned}
!_n A &\multimap 1 \\
!_{1+n} A &\multimap A \\
!_{n+m} A &\multimap !_n A \otimes !_m A \\
!_{nm} A &\multimap !_n !_m A
\end{aligned}
$$

where n and m are natural numbers.

Bounded Linear Logic.

- It is a refinement on (intuitionistic) linear logic:

$$\begin{aligned} !_n A &\multimap 1 \\ !_{1+n} A &\multimap A \\ !_{n+m} A &\multimap !_n A \otimes !_m A \\ !_{nm} A &\multimap !_n !_m A \end{aligned}$$

  where n and m are natural numbers.
- More generally, n ad m could be polynomials (on possibly many variables) and not just natural numbers.
- Moreover, ! can act as a binder for resource variables: $!_{x<p}A$. As an example, the following is an axiom:

$$!_{x<p+q}A \multimap !_{x<p}A \otimes !_{y<q}A[x \leftarrow y+p]$$

- Intuitively:

$$!_{x<p}A \sim A[x \leftarrow 0] \otimes A[x \leftarrow 1] \otimes \ldots \otimes A[x \leftarrow p-1].$$

Quantified Bounded Affine Logic.

- Usual, second order quantification is available in QBAL.
  - It was available in BLL, too.
- There is another form of quantification in QBAL: universal and existential quantification on resource variables:

$$\exists(\overline{x}) : \mathcal{C}.A$$
$$\forall(\overline{x}) : \mathcal{C}.A$$

  where $\mathcal{C}$ is a set of constraints.
- Example:

$$\exists(x, y) : \{x \leq z^2, y \leq x\}.!_{xy^2} A$$

  Notice that the constraints in $\{x \leq z^2, y \leq x\}$ enforce a polynomial upper bound on both x and y: $x \leq z^2$ and $y \leq x \leq z^2$.
  - Not by coincidence! This is a constraint.
- Sequents have the form $\Gamma \vdash_{\mathcal{C}} A$.
- Rules for first-order quantifier are standard.
- The rules coming from BLL leaves $\mathcal{C}$ unchanged.

Rules: Some Examples

- Axiom:

$$\frac{A \sqsubseteq_{\mathcal{C}} B}{A \vdash_{\mathcal{C}} B}\ A$$

Rules: Some Examples

- Axiom:

$$\frac{A \sqsubseteq_{\mathcal{C}} B}{A \vdash_{\mathcal{C}} B}\ A$$

- Linear arrow:

$$\frac{\Gamma \vdash_{\mathcal{C}} A \quad \Delta, B \vdash_{\mathcal{C}} C}{\Gamma, \Delta, A \multimap B \vdash_{\mathcal{C}} C}\ L_{\multimap} \qquad \frac{\Gamma, A \vdash_{\mathcal{C}} B}{\Gamma \vdash_{\mathcal{C}} A \multimap B}\ R_{\multimap}$$

Rules: Some Examples

- Axiom:

$$\frac{A \sqsubseteq_{\mathcal{C}} B}{A \vdash_{\mathcal{C}} B}\ A$$

- Linear arrow:

$$\frac{\Gamma \vdash_{\mathcal{C}} A \quad \Delta, B \vdash_{\mathcal{C}} C}{\Gamma, \Delta, A \multimap B \vdash_{\mathcal{C}} C}\ L_{\multimap} \qquad \frac{\Gamma, A \vdash_{\mathcal{C}} B}{\Gamma \vdash_{\mathcal{C}} A \multimap B}\ R_{\multimap}$$

- Promotion:

$$\frac{A_1, \ldots, A_n \vdash_{\mathcal{C}} B \quad \mathcal{D}, x < p \models \mathcal{C} \quad x \notin FV(\mathcal{D}) \quad p \sqsubseteq_{\mathcal{D}} q_i}{!_{x<q_1} A_1, \ldots, !_{x<q_n} A_n \vdash_{\mathcal{D}} !_{x<p} B}\ P_!$$

Rules: Some Examples

- Axiom:

$$\frac{A \sqsubseteq_{\mathcal{C}} B}{A \vdash_{\mathcal{C}} B}\ A$$

- Linear arrow:

$$\frac{\Gamma \vdash_{\mathcal{C}} A \quad \Delta, B \vdash_{\mathcal{C}} C}{\Gamma, \Delta, A \multimap B \vdash_{\mathcal{C}} C}\ L_{\multimap} \qquad \frac{\Gamma, A \vdash_{\mathcal{C}} B}{\Gamma \vdash_{\mathcal{C}} A \multimap B}\ R_{\multimap}$$

- Promotion:

$$\frac{A_1, \ldots, A_n \vdash_{\mathcal{C}} B \quad \mathcal{D}, x < p \models \mathcal{C} \quad x \notin FV(\mathcal{D}) \quad p \sqsubseteq_{\mathcal{D}} q_i}{!_{x<q_1} A_1, \ldots, !_{x<q_n} A_n \vdash_{\mathcal{D}} !_{x<p} B}\ P_!$$

- Existential quantification:

$$\frac{\Gamma \vdash_{\mathcal{C}} A\{\overline{p}/\overline{x}\} \quad \mathcal{C} \models \mathcal{D}\{\overline{p}/\overline{x}\}}{\Gamma \vdash_{\mathcal{C}} \exists \overline{x} : \mathcal{D}.A}\ R_{\exists x}$$

$$\frac{\Gamma, A \vdash_{\mathcal{C} \cup \mathcal{D}} C \quad \overline{x} \notin FV(\Gamma) \cup FV(C) \cup FV(\mathcal{C})}{\Gamma, \exists \overline{x} : \mathcal{D} : A \vdash_{\mathcal{C}} C}\ L_{\exists x}$$

Programming in QBAL...

- ...is not so different from programming in intuitionistic second-order logic.
- For example, the type of natural numbers has the same structure:

  $$\mathbf{N}_p = \forall\alpha.!_{x<p}(\alpha(x) \multimap \alpha(x+1)) \multimap \alpha(0) \multimap \alpha(p)$$

  It is, however, parametrized on a polynomial p.
- Similarly for any word algebra W.
- These types support the usual impredicative iteration schema.
- The added value provided by first-order quantification will show up in the embeddings.

How Intentionally Expressive QBAL is?

How Intentionally Expressive QBAL is?

Just look at which ICC systems
can be embeddded into it. . .

How Intentionally Expressive QBAL is?

Just look at which ICC systems
can be embeddded into it. . .

. . . (**compositionally**).

Embedding LFPL.

- LFPL is a linear functional language [Hofmann1999].

Embedding LFPL.

- LFPL is a linear functional language [Hofmann1999].
  - All functions are non-size-increasing by construction.
  - Linear higher-order primitive recursion.
  - Nested recursion allowed.
  - Types: $A, B ::= \diamond \mid Nat \mid A \otimes B \mid A \multimap B$.

Embedding LFPL.

- LFPL is a linear functional language [Hofmann1999].
  - All functions are non-size-increasing by construction.
  - Linear higher-order primitive recursion.
  - Nested recursion allowed.
  - Types: $A, B ::= \diamond \mid Nat \mid A \otimes B \mid A \multimap B$.
- The embedding:

$$\begin{aligned}
\langle \diamond \rangle_p^q &= \exists \varepsilon : \{1 \leq p\}.1 \\
\langle Nat \rangle_p^q &= \mathbf{N}_p \\
\langle A \otimes B \rangle_p^q &= \exists (x, y) : \{x + y \leq p\}.\langle A \rangle_x^q \otimes \langle B \rangle_y^q \\
\langle A \multimap B \rangle_p^q &= \forall (x) : \{x + p \leq q\}.\langle A \rangle_x^q \multimap \langle B \rangle_{x+p}^q
\end{aligned}$$

Embedding LFPL.

- LFPL is a linear functional language [Hofmann1999].
  - All functions are non-size-increasing by construction.
  - Linear higher-order primitive recursion.
  - Nested recursion allowed.
  - Types: $A, B ::= \diamond \mid Nat \mid A \otimes B \mid A \multimap B$.
- The embedding:

$$\begin{aligned}
\langle \diamond \rangle_p^q &= \exists \varepsilon : \{1 \leq p\}.1 \\
\langle Nat \rangle_p^q &= \mathbf{N}_p \\
\langle A \otimes B \rangle_p^q &= \exists (x, y) : \{x + y \leq p\}.\langle A \rangle_x^q \otimes \langle B \rangle_y^q \\
\langle A \multimap B \rangle_p^q &= \forall (x) : \{x + p \leq q\}.\langle A \rangle_x^q \multimap \langle B \rangle_{x+p}^q
\end{aligned}$$

and

$$x : A_1, \ldots, x : A_n \vdash M : B$$

Embedding LFPL.

- LFPL is a linear functional language [Hofmann1999].
  - All functions are non-size-increasing by construction.
  - Linear higher-order primitive recursion.
  - Nested recursion allowed.
  - Types: $A, B ::= \diamond \mid Nat \mid A \otimes B \mid A \multimap B$.
- The embedding:

$$\begin{aligned}
\langle \diamond \rangle_p^q &= \exists \varepsilon : \{1 \leq p\}.1 \\
\langle Nat \rangle_p^q &= \mathbf{N}_p \\
\langle A \otimes B \rangle_p^q &= \exists (x, y) : \{x + y \leq p\}.\langle A \rangle_x^q \otimes \langle B \rangle_y^q \\
\langle A \multimap B \rangle_p^q &= \forall (x) : \{x + p \leq q\}.\langle A \rangle_x^q \multimap \langle B \rangle_{x+p}^q
\end{aligned}$$

and

$$x : A_1, \ldots, x : A_n \vdash M : B$$
$$\Downarrow$$

Embedding LFPL.

- LFPL is a linear functional language [Hofmann1999].
  - All functions are non-size-increasing by construction.
  - Linear higher-order primitive recursion.
  - Nested recursion allowed.
  - Types: $A, B ::= \diamond \mid Nat \mid A \otimes B \mid A \multimap B$.
- The embedding:

$$\begin{aligned}
\langle \diamond \rangle_p^q &= \exists \varepsilon : \{1 \leq p\}.1 \\
\langle Nat \rangle_p^q &= \mathbf{N}_p \\
\langle A \otimes B \rangle_p^q &= \exists (x, y) : \{x + y \leq p\}.\langle A \rangle_x^q \otimes \langle B \rangle_y^q \\
\langle A \multimap B \rangle_p^q &= \forall (x) : \{x + p \leq q\}.\langle A \rangle_x^q \multimap \langle B \rangle_{x+p}^q
\end{aligned}$$

and

$$x : A_1, \ldots, x : A_n \vdash M : B$$
$$\Downarrow$$
$$\langle A_1 \rangle_{x_1}^{\sum_{i=1}^n x_i}, \ldots, \langle A_n \rangle_{x_n}^{\sum_{i=1}^n x_i} \vdash_\emptyset \langle B \rangle_{\sum_{i=1}^n x_i}^{\sum_{i=1}^n x_i}$$

Embedding BC.

- BC is a characterization of the polytime functions introduced in the nineties [BC93].
  - A subalgebra of the primitive recursion functions.
  - Every function $f : \mathbb{B}^n \to \mathbb{B}$ has m normal arguments and $n - m$ safe arguments.
  - In a recursive definition, the recursive call must go through a safe argument, while the argument driving the recursion must be normal.

Embedding BC.

- BC is a characterization of the polytime functions introduced in the nineties [BC93].
  - A subalgebra of the primitive recursion functions.
  - Every function $f : \mathbb{B}^n \to \mathbb{B}$ has m normal arguments and $n - m$ safe arguments.
  - In a recursive definition, the recursive call must go through a safe argument, while the argument driving the recursion must be normal.
- The embedding:

$$f : \mathbb{B} \times \ldots \times \mathbb{B} \to \mathbb{B} \in \mathsf{BC}$$

Embedding BC.

- BC is a characterization of the polytime functions introduced in the nineties [BC93].
  - A subalgebra of the primitive recursion functions.
  - Every function $f : \mathbb{B}^n \to \mathbb{B}$ has m normal arguments and $n - m$ safe arguments.
  - In a recursive definition, the recursive call must go through a safe argument, while the argument driving the recursion must be normal.
- The embedding:

$$f : \mathbb{B} \times \ldots \times \mathbb{B} \to \mathbb{B} \in \mathsf{BC}$$
$$\Downarrow$$

Embedding BC.

- BC is a characterization of the polytime functions introduced in the nineties [BC93].
  - A subalgebra of the primitive recursion functions.
  - Every function $f : \mathbb{B}^n \to \mathbb{B}$ has m normal arguments and $n - m$ safe arguments.
  - In a recursive definition, the recursive call must go through a safe argument, while the argument driving the recursion must be normal.
- The embedding:

$$f : \mathbb{B} \times \ldots \times \mathbb{B} \to \mathbb{B} \in \mathsf{BC}$$

$$\Downarrow$$

$$\mathbf{W}_{x_1}, \ldots, \mathbf{W}_{x_n} \vdash_{x_{m+1} \leq y, \ldots, x_n \leq y} \mathbf{W}_{p(x_1, \ldots, x_m)+y}$$

Embedding BC.

- BC is a characterization of the polytime functions introduced in the nineties [BC93].
  - A subalgebra of the primitive recursion functions.
  - Every function $f : \mathbb{B}^n \to \mathbb{B}$ has m normal arguments and $n - m$ safe arguments.
  - In a recursive definition, the recursive call must go through a safe argument, while the argument driving the recursion must be normal.
- The embedding:

$$\begin{array}{c} f : \mathbb{B} \times \ldots \times \mathbb{B} \to \mathbb{B} \in \mathsf{BC} \\ \Downarrow \\ \mathbf{W}_{x_1}, \ldots, \mathbf{W}_{x_n} \vdash_{x_{m+1} \leq y, \ldots, x_n \leq y} \mathbf{W}_{p(x_1, \ldots, x_m) + y} \end{array}$$

- This is an induction on the structure of the proof that f is a BC function..
- Interestingly, the embedding is very similar to the proof of soundness for BC.

Embedding BC.

- BC is a characterization of the polytime functions introduced in the nineties [BC93].
  - A subalgebra of the primitive recursion functions.
  - Every function $f : \mathbb{B}^n \to \mathbb{B}$ has m normal arguments and $n - m$ safe arguments.
  - In a recursive definition, the recursive call must go through a safe argument, while the argument driving the recursion must be normal.
- The embedding:

$$f : \mathbb{B} \times \ldots \times \mathbb{B} \to \mathbb{B} \in \mathsf{BC}$$
$$\Downarrow$$
$$\mathbf{W}_{x_1}, \ldots, \mathbf{W}_{x_n} \vdash_{x_{m+1} \leq y, \ldots, x_n \leq y} \mathbf{W}_{p(x_1, \ldots, x_m) + y}$$

- This is an induction on the structure of the proof that f is a BC function..
- Interestingly, the embedding is very similar to the proof of soundness for BC.
- This cannot be done in ordinary BLL.

What about Light Logics?

- Here the situation is different.

$$
\begin{gathered}
!A \multimap 1 \\
!A \multimap A \\
!A \multimap !A \otimes !A \\
!A \multimap !!A
\end{gathered}
$$

What about Light Logics?

- Here the situation is different.

$$!A \multimap 1$$

$$!A \multimap !A \otimes !A$$

What about Light Logics?

- Here the situation is different.

$$!A \multimap 1$$

$$!A \multimap !A \otimes !A$$

- It is impossible to embed any light logic (except SLL) into QBAL.

What about Light Logics?

- Here the situation is different.

$$!A \multimap 1$$

$$!A \multimap !A \otimes !A$$

- It is impossible to embed any light logic (except SLL) into QBAL.
- Actually, ELL can be embedded into (Q)BAL.

$$!A$$
$$\Downarrow$$
$$!_{x<0}A$$

What about Light Logics?

- Here the situation is different.

$$!A \multimap 1$$

$$!A \multimap !A \otimes !A$$

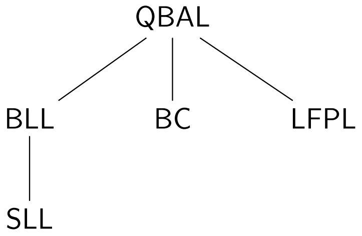
- It is impossible to embed any light logic (except SLL) into QBAL.
- Actually, ELL can be embedded into (Q)BAL.

$$\begin{gathered} !A \\ \Downarrow \\ !_{x<0}A \end{gathered}$$

- But the embedding is not sensible from a dynamical point of view!

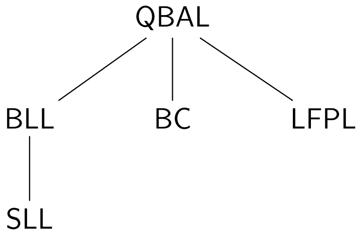
QBAL: Summing Up

- **Moral**: (a natural extension of) BLL is an interesting ICC system with strong intensional expressive power:



QBAL: Summing Up

- **Moral**: (a natural extension of) BLL is an interesting ICC system with strong intensional expressive power:



- **Price to pay**: the system is not implicit!

Intensional Completeness?

- **Extensional Completeness**
  - The set of all functions computed by programs in $\mathcal{S}$ *equals* the complexity class $\mathcal{P}$
  - $\mathcal{S}$, seen as a language, can have a very low complexity, even polynomial time in some cases.
  - $\mathcal{S}$ can be a *tiny subset* of $\mathcal{L}_{\mathcal{S}}$, thus practically useless for complexity analysis.
  - Many examples: function algebras, light logics, etc.

Intensional Completeness?

- **Extensional Completeness**
  - The set of all functions computed by programs in $\mathcal{S}$ *equals* the complexity class $\mathcal{P}$
  - $\mathcal{S}$, seen as a language, can have a very low complexity, even polynomial time in some cases.
  - $\mathcal{S}$ can be a *tiny subset* of $\mathcal{L}_{\mathcal{S}}$, thus practically useless for complexity analysis.
  - Many examples: function algebras, light logics, etc.
- **Intensional Completeness**
  - The set $\mathcal{L}_{\mathcal{P}}$ of all efficient programs *equals* $\mathcal{S}$.
  - $\mathcal{S}$, seen as a language, cannot be recursively enumerable.
  - Added value: a bound on the complexity of $P \in \mathcal{S}$ can often be read off from the proof of P being an element of $\mathcal{S}$.
  - Few examples: $\mathsf{d}\ell\mathsf{PCF}$.

Part IV

Challenges: Cryptography and Concurrency

Proofs by Reduction

$$\mathbf{Property}(\Phi) \overset{def}{=} (\forall \mathcal{D}. PPT(\mathcal{D}) \Rightarrow \mathsf{Pr}(Success(\mathcal{D} \mid \Phi)) = negl(n))\,.$$

Proofs by Reduction

$$\textbf{Property}(\Phi) \overset{def}{=} (\forall \mathcal{D}.PPT(\mathcal{D}) \Rightarrow \mathsf{Pr}(Success(\mathcal{D} \mid \Phi)) = negl(n))\,.$$

$$\textbf{Assumption}(\Phi) \quad \Rightarrow \quad \textbf{Security}(\Pi)$$

Proofs by Reduction

$$\mathbf{Property}(\Phi) \overset{def}{=} (\forall \mathcal{D}. PPT(\mathcal{D}) \Rightarrow \mathsf{Pr}(Success(\mathcal{D} \mid \Phi)) = negl(n))\,.$$

$$\mathbf{Assumption}(\Phi) \quad \Rightarrow \quad \mathbf{Security}(\Pi)$$

Examples

Proofs by Reduction

$$\textbf{Property}(\Phi) \overset{def}{=} (\forall \mathcal{D}. PPT(\mathcal{D}) \Rightarrow \mathsf{Pr}(Success(\mathcal{D} \mid \Phi)) = negl(n))\,.$$

$$\textbf{Assumption}(\Phi) \quad \Rightarrow \quad \textbf{Security}(\Pi)$$

Examples

$$\textbf{OneWay}(f) \quad \Rightarrow \quad \textbf{PRG}(G)$$

Proofs by Reduction

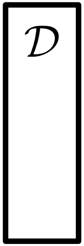
$$\textbf{Property}(\Phi) \overset{def}{=} (\forall \mathcal{D}.PPT(\mathcal{D}) \Rightarrow \mathsf{Pr}(Success(\mathcal{D} \mid \Phi)) = negl(n))\,.$$

$$\textbf{Assumption}(\Phi) \quad \Rightarrow \quad \textbf{Security}(\Pi)$$

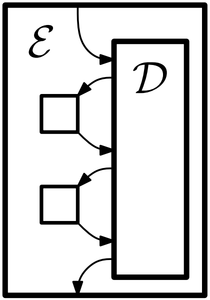
Examples

$$\textbf{OneWay}(f) \quad \Rightarrow \quad \textbf{PRG}(G)$$
$$\textbf{PRG}(G) \quad \Rightarrow \quad \textbf{IND}(\Pi)$$

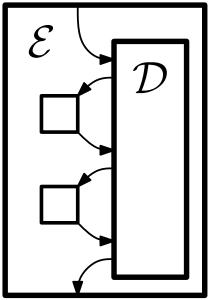
Proofs by Reduction

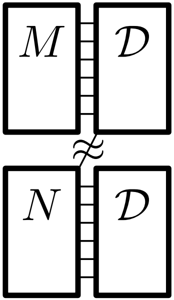


Proofs by Reduction

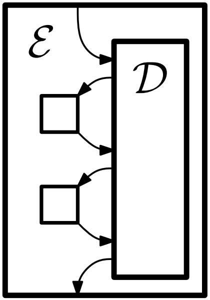


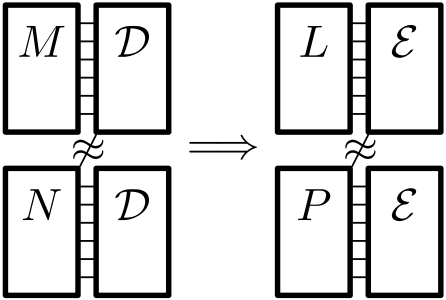
Proofs by Reduction



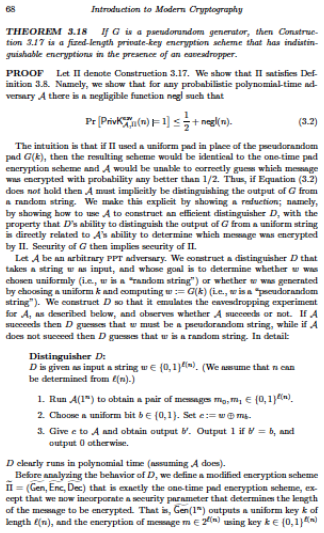


Proofs by Reduction





An Example



THEOREM 3.18 *If G is a pseudorandom generator, then Construction 3.17 is a fixed-length private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper.*

PROOF Let Π denote Construction 3.17. We show that Π satisfies Definition 3.8. Namely, we show that for any probabilistic polynomial-time adversary $\mathcal{A}$ there is a negligible function negl such that

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) \models 1\right] \leq \frac{1}{2} + \mathsf{negl}(n). \qquad (3.2)$$

The intuition is that if Π used a uniform pad in place of the pseudorandom pad $G(k)$, then the resulting scheme would be identical to the one-time pad encryption scheme and $\mathcal{A}$ would be unable to correctly guess which message was encrypted with probability any better than $1/2$. Thus, if Equation (3.2) does *not* hold then $\mathcal{A}$ must implicitly be distinguishing the output of G from a random string. We make this explicit by showing a *reduction*; namely, by showing how to use $\mathcal{A}$ to construct an efficient distinguisher D, with the property that D's ability to distinguish the output of G from a uniform string is directly related to $\mathcal{A}$'s ability to determine which message was encrypted by Π. Security of G then implies security of Π.

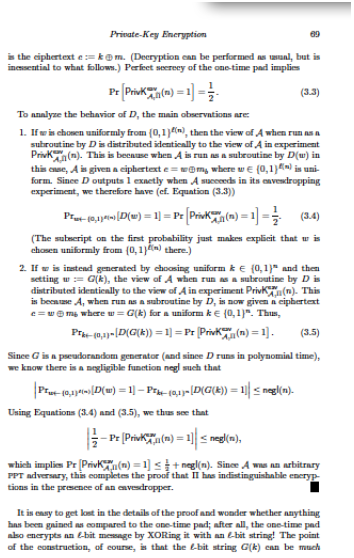
Let $\mathcal{A}$ be an arbitrary PPT adversary. We construct a distinguisher D that takes a string w as input, and whose goal is to determine whether w was chosen uniformly (i.e., w is a "random string") or whether w was generated by choosing a uniform k and computing $w := G(k)$ (i.e., w is a "pseudorandom string"). We construct D so that it emulates the eavesdropping experiment for $\mathcal{A}$, as described below, and observes whether $\mathcal{A}$ succeeds or not. If $\mathcal{A}$ succeeds then D guesses that w must be a pseudorandom string, while if $\mathcal{A}$ does not succeed then D guesses that w is a random string. In detail:

Distinguisher D:
D is given as input a string $w \in \{0,1\}^{\ell(n)}$. (We assume that n can be determined from $\ell(n)$.)

1. Run $\mathcal{A}(1^n)$ to obtain a pair of messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$.
2. Choose a uniform bit $b \in \{0,1\}$. Set $c := w \oplus m_b$.
3. Give c to $\mathcal{A}$ and obtain output b'. Output 1 if $b' = b$, and output 0 otherwise.

D clearly runs in polynomial time (assuming $\mathcal{A}$ does).

Before analyzing the behavior of D, we define a modified encryption scheme $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ that is exactly the one-time pad encryption scheme, except that we now incorporate a security parameter that determines the length of the message to be encrypted. That is, $\widetilde{\mathsf{Gen}}(1^n)$ outputs a uniform key k of length $\ell(n)$, and the encryption of message $m \in 2^{\ell(n)}$ using key $k \in \{0,1\}^{\ell(n)}$



is the ciphertext $c := k \oplus m$. (Decryption can be performed as usual, but is inessential to what follows.) Perfect secrecy of the one-time pad implies

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] = \frac{1}{2}. \qquad (3.3)$$

To analyze the behavior of D, the main observations are:

1. If w is chosen uniformly from $\{0,1\}^{\ell(n)}$, then the view of $\mathcal{A}$ when run as a subroutine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n)$. This is because when $\mathcal{A}$ is run as a subroutine by $D(w)$ in this case, $\mathcal{A}$ is given a ciphertext $c = w \oplus m_b$ where $w \in \{0,1\}^{\ell(n)}$ is uniform. Since D outputs 1 exactly when $\mathcal{A}$ succeeds in its eavesdropping experiment, we therefore have (cf. Equation (3.3))
$$\Pr_{w \leftarrow \{0,1\}^{\ell(n)}}[D(w) = 1] = \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] = \frac{1}{2}. \qquad (3.4)$$
(The subscript on the first probability just makes explicit that w is chosen uniformly from $\{0,1\}^{\ell(n)}$ there.)
2. If w is instead generated by choosing uniform $k \in \{0,1\}^n$ and then setting $w := G(k)$, the view of $\mathcal{A}$ when run as a subroutine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. This is because $\mathcal{A}$, when run as a subroutine by D, is now given a ciphertext $c = w \oplus m_b$ where $w = G(k)$ for a uniform $k \in \{0,1\}^n$. Thus,
$$\Pr_{k \leftarrow \{0,1\}^n}[D(G(k)) = 1] = \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right]. \qquad (3.5)$$

Since G is a pseudorandom generator (and since D runs in polynomial time), we know there is a negligible function negl such that

$$\left|\Pr_{w \leftarrow \{0,1\}^{\ell(n)}}[D(w) = 1] - \Pr_{k \leftarrow \{0,1\}^n}[D(G(k)) = 1]\right| \leq \mathsf{negl}(n).$$

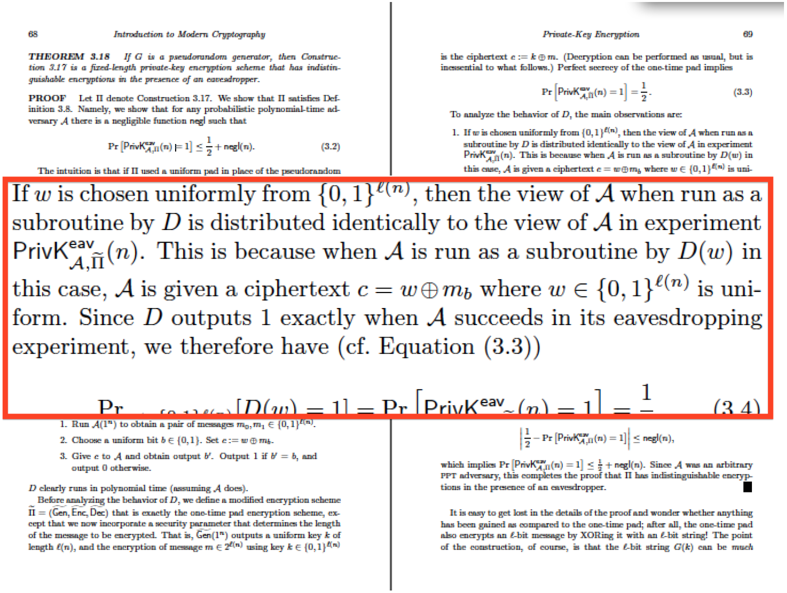
Using Equations (3.4) and (3.5), we thus see that

$$\left|\frac{1}{2} - \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right]\right| \leq \mathsf{negl}(n),$$

which implies $\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n)$. Since $\mathcal{A}$ was an arbitrary PPT adversary, this completes the proof that Π has indistinguishable encryptions in the presence of an eavesdropper. $\blacksquare$

It is easy to get lost in the details of the proof and wonder whether anything has been gained as compared to the one-time pad; after all, the one-time pad also encrypts an ℓ-bit message by XORing it with an ℓ-bit string! The point of the construction, of course, is that the ℓ-bit string $G(k)$ can be *much*

An Example



THEOREM 3.18 *If G is a pseudorandom generator, then Construction 3.17 is a fixed-length private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper.*

PROOF Let Π denote Construction 3.17. We show that Π satisfies Definition 3.8. Namely, we show that for any probabilistic polynomial-time adversary $\mathcal{A}$ there is a negligible function negl such that

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) \models 1\right] \leq \frac{1}{2} + \mathsf{negl}(n). \qquad (3.2)$$

The intuition is that if Π used a uniform pad in place of the pseudorandom

If w is chosen uniformly from $\{0,1\}^{\ell(n)}$, then the view of $\mathcal{A}$ when run as a subroutine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n)$. This is because when $\mathcal{A}$ is run as a subroutine by $D(w)$ in this case, $\mathcal{A}$ is given a ciphertext $c = w \oplus m_b$ where $w \in \{0,1\}^{\ell(n)}$ is uniform. Since D outputs 1 exactly when $\mathcal{A}$ succeeds in its eavesdropping experiment, we therefore have (cf. Equation (3.3))

$$\Pr_{\{0,1\}^{\ell(n)}}[D(w)=1] = \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\sim}(n)=1\right] = \frac{1}{} \qquad (3.4)$$

1. Run $\mathcal{A}(1^n)$ to obtain a pair of messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$.
2. Choose a uniform bit $b \in \{0,1\}$. Set $c := w \oplus m_b$.
3. Give c to $\mathcal{A}$ and obtain output b'. Output 1 if $b' = b$, and output 0 otherwise.

D clearly runs in polynomial time (assuming $\mathcal{A}$ does).

Before analyzing the behavior of D, we define a modified encryption scheme $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ that is exactly the one-time pad encryption scheme, except that we now incorporate a security parameter that determines the length of the message to be encrypted. That is, $\widetilde{\mathsf{Gen}}(1^n)$ outputs a uniform key k of length $\ell(n)$, and the encryption of message $m \in 2^{\ell(n)}$ using key $k \in \{0,1\}^{\ell(n)}$

is the ciphertext $c := k \oplus m$. (Decryption can be performed as usual, but is inessential to what follows.) Perfect secrecy of the one-time pad implies

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] = \frac{1}{2}. \qquad (3.3)$$

To analyze the behavior of D, the main observations are:

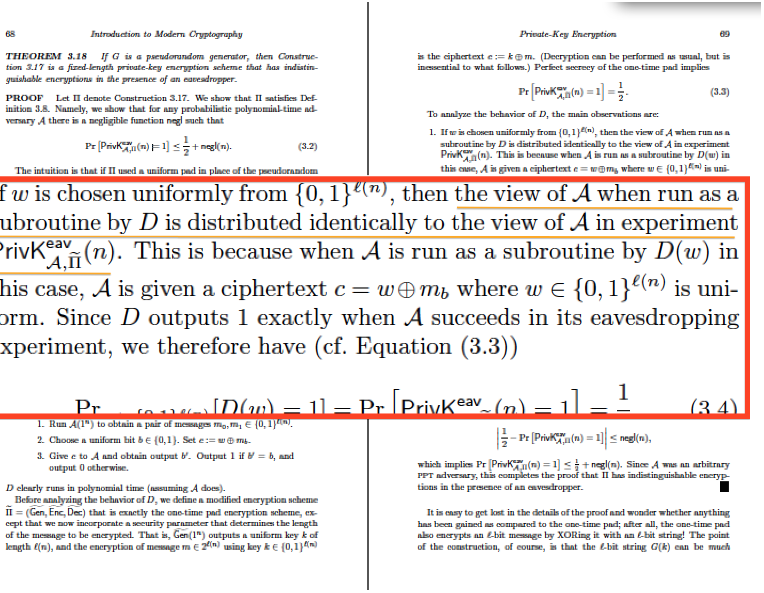
1. If w is chosen uniformly from $\{0,1\}^{\ell(n)}$, then the view of $\mathcal{A}$ when run as a subroutine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n)$. This is because when $\mathcal{A}$ is run as a subroutine by $D(w)$ in this case, $\mathcal{A}$ is given a ciphertext $c = w \oplus m_b$ where $w \in \{0,1\}^{\ell(n)}$ is uni-

$$\left|\frac{1}{2} - \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right]\right| \leq \mathsf{negl}(n),$$

which implies $\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n)$. Since $\mathcal{A}$ was an arbitrary PPT adversary, this completes the proof that Π has indistinguishable encryptions in the presence of an eavesdropper. ■

It is easy to get lost in the details of the proof and wonder whether anything has been gained as compared to the one-time pad; after all, the one-time pad also encrypts an ℓ-bit message by XORing it with an ℓ-bit string! The point of the construction, of course, is that the ℓ-bit string $G(k)$ can be *much*

An Example



THEOREM 3.18 *If G is a pseudorandom generator, then Construction 3.17 is a fixed-length private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper.*

PROOF Let Π denote Construction 3.17. We show that Π satisfies Definition 3.8. Namely, we show that for any probabilistic polynomial-time adversary $\mathcal{A}$ there is a negligible function negl such that

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) \models 1\right] \leq \frac{1}{2} + \mathsf{negl}(n). \qquad (3.2)$$

The intuition is that if Π used a uniform pad in place of the pseudorandom

1. Run $\mathcal{A}(1^n)$ to obtain a pair of messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$.
2. Choose a uniform bit $b \in \{0,1\}$. Set $c := w \oplus m_b$.
3. Give c to $\mathcal{A}$ and obtain output b'. Output 1 if $b' = b$, and output 0 otherwise.

D clearly runs in polynomial time (assuming $\mathcal{A}$ does).

Before analyzing the behavior of D, we define a modified encryption scheme $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ that is exactly the one-time pad encryption scheme, except that we now incorporate a security parameter that determines the length of the message to be encrypted. That is, $\widetilde{\mathsf{Gen}}(1^n)$ outputs a uniform key k of length $\ell(n)$, and the encryption of message $m \in 2^{\ell(n)}$ using key $k \in \{0,1\}^{\ell(n)}$

is the ciphertext $c := k \oplus m$. (Decryption can be performed as usual, but is inessential to what follows.) Perfect secrecy of the one-time pad implies

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] = \frac{1}{2}. \qquad (3.3)$$

To analyze the behavior of D, the main observations are:

1. If w is chosen uniformly from $\{0,1\}^{\ell(n)}$, then the view of $\mathcal{A}$ when run as a subroutine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n)$. This is because when $\mathcal{A}$ is run as a subroutine by $D(w)$ in this case, $\mathcal{A}$ is given a ciphertext $c = w \oplus m_b$ where $w \in \{0,1\}^{\ell(n)}$ is uni-

If w is chosen uniformly from $\{0,1\}^{\ell(n)}$, then the view of $\mathcal{A}$ when run as a subroutine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n)$. This is because when $\mathcal{A}$ is run as a subroutine by $D(w)$ in this case, $\mathcal{A}$ is given a ciphertext $c = w \oplus m_b$ where $w \in \{0,1\}^{\ell(n)}$ is uniform. Since D outputs 1 exactly when $\mathcal{A}$ succeeds in its eavesdropping experiment, we therefore have (cf. Equation (3.3))

$$\Pr_{w \leftarrow \{0,1\}^{\ell(n)}}[D(w) = 1] = \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\widetilde{\Pi}}(n) = 1\right] = \frac{1}{} \qquad (3.4)$$

$$\left|\frac{1}{2} - \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right]\right| \leq \mathsf{negl}(n),$$

which implies $\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n)$. Since $\mathcal{A}$ was an arbitrary PPT adversary, this completes the proof that Π has indistinguishable encryptions in the presence of an eavesdropper. ■

It is easy to get lost in the details of the proof and wonder whether anything has been gained as compared to the one-time pad; after all, the one-time pad also encrypts an ℓ-bit message by XORing it with an ℓ-bit string! The point of the construction, of course, is that the ℓ-bit string $G(k)$ can be *much*

An Alternative Viewpoint

1. Introduce a λ-calculus RSLR, and prove it **complete** for probabilistic polynomial-time computation [Zhang10,DLPT11].
    - RSLR is nothing more than a randomized variation on Hofmann's SLR obtained by endowing the latter with a primitive `flip`.

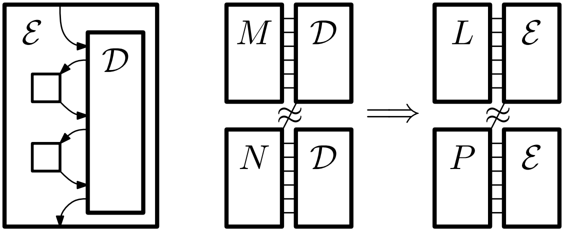
An Alternative Viewpoint

1. Introduce a λ-calculus RSLR, and prove it **complete** for probabilistic polynomial-time computation [Zhang10,DLPT11].
   - RSLR is nothing more than a randomized variation on Hofmann's SLR obtained by endowing the latter with a primitive `flip`.
2. Define a notion of **equivalence** close enough to *computational indistinguishability*.
   - Two distribution ensembles $\{\mathcal{D}_n\}$ and $\{\mathcal{E}_n\}$ are computational indistinguishable when no PPT algorithm can distinguish between them with more than a negligible probability of success.

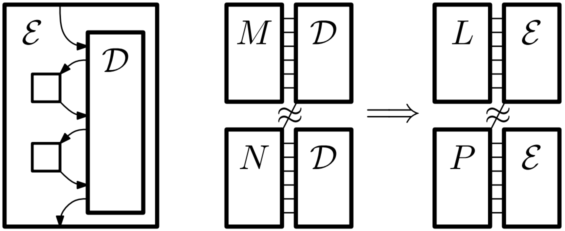
An Alternative Viewpoint

1. Introduce a λ-calculus RSLR, and prove it **complete** for probabilistic polynomial-time computation [Zhang10,DLPT11].
   - RSLR is nothing more than a randomized variation on Hofmann's SLR obtained by endowing the latter with a primitive `flip`.
2. Define a notion of **equivalence** close enough to *computational indistinguishability*.
   - Two distribution ensembles $\{\mathcal{D}_n\}$ and $\{\mathcal{E}_n\}$ are computational indistinguishable when no PPT algorithm can distinguish between them with more than a negligible probability of success.
3. Prove an appropriate **context lemma** in the form of full-abstraction for trace equivalence.

Cryptographic Reductions, More Formally

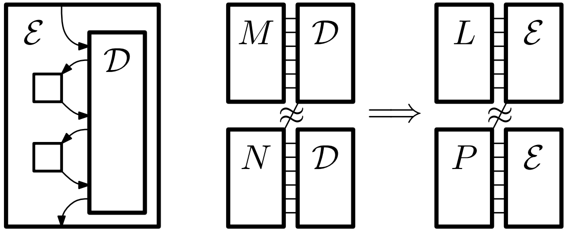


Cryptographic Reductions, More Formally



- From the fact that $M \not\equiv N$...

Cryptographic Reductions, More Formally



- From the fact that $M \not\equiv N$…
- …one derives the existence of a context $\mathcal{D}$ separating M and N…
- …from which, thanks to full abstraction, one derives the existence of a trace T which separates M and N…
- …from which one builds a trace S separating L and P.

Probabilistic ICC: Challenges

- Besides applications to cryptographic proofs, there is also the question of better understanding probabilistic complexity classes like ZPP or BPP.

Probabilistic ICC: Challenges

- Besides applications to cryptographic proofs, there is also the question of better understanding probabilistic complexity classes like ZPP or BPP.
- There is however an inherent difficulty in pursuing this goal!
    - Classes like ZPP or BPP, as opposed to P, NP, PSPACE, etc. which are *syntactic*, are *semantic*: there is no easy way to enumerate even the **machines** which define the class.
    - This is due to the fact that "BPP-machines" are required to produce the correct output only with a certain probability, and checking so is not trivial.
    - But how about characterizing (interesting) subclasses of them? This is an excellent research topic!

ICC and Concurrency?

- The computational models we have considered so far are *sequential*. How about *concurrent* models of computation?

ICC and Concurrency?

- The computational models we have considered so far are *sequential*. How about *concurrent* models of computation?
- The prototypical example of such a model of computation is Milner's π-calculus, in which so-called *processes* take the place of *terms*.
  - A π-calculus process can be seen as modeling a systems of n threads $T_1, \ldots, T_n$ which run in parallel.
  - The threads can interact by exchanging messages through channels, which themselves can contain addresses of other channels.

ICC and Concurrency?

- The computational models we have considered so far are *sequential*. How about *concurrent* models of computation?
- The prototypical example of such a model of computation is Milner's π-calculus, in which so-called *processes* take the place of *terms*.
  - A π-calculus process can be seen as modeling a systems of n threads $T_1, \ldots, T_n$ which run in parallel.
  - The threads can interact by exchanging messages through channels, which themselves can contain addresses of other channels.
- Exactly as for programs/terms/algorithms, we would like processes to be efficient.
  - But what does it mean?

Higher-Order π-Calculus

- Processes:

$$V ::= \star \mid \lambda x.P$$
$$P ::= \mathbf{0} \mid x \mid P \parallel P \mid a\langle x\rangle.P \mid \overline{a}\langle V\rangle.P \mid (\nu a)P \mid VV$$

- Reduction:

$$\frac{}{\overline{a}\langle V\rangle.P \parallel a\langle x\rangle.Q \rightarrow_{\mathsf{P}} P \parallel Q[x/V]} \qquad \frac{}{(\lambda x.P)V \rightarrow_{\mathsf{P}} P[x/V]}$$

$$\frac{P \rightarrow_{\mathsf{P}} Q}{P \parallel R \rightarrow_{\mathsf{P}} Q \parallel R} \qquad \frac{P \rightarrow_{\mathsf{P}} Q}{(\nu a)P \rightarrow_{\mathsf{P}} (\nu a)Q}$$

$$\frac{P \equiv Q \quad Q \rightarrow_{\mathsf{P}} R \quad R \equiv S}{P \rightarrow_{\mathsf{P}} S}$$

Higher-Order π-Calculus

- Nontermination:

$$V = \lambda y.a\langle x\rangle.(x\star \parallel \overline{a}\langle x\rangle)$$
$$Q = V\star \parallel \overline{a}\langle V\rangle$$

Indeed:

$$Q \to Q \to \dots$$

- More interesting example:

$$V = \lambda z.a\langle x\rangle.(b\langle y\rangle.\overline{c}\langle y\rangle.x\star \parallel \overline{a}\langle x\rangle)$$
$$Q = V\star \parallel \overline{a}\langle V\rangle$$

Linear Higher-Order π-Calculus: $\mathsf{LHO}\pi$

- Generalizing $\Lambda \Longrightarrow \Lambda_!$ to processes.
- Values and Processes:

$$V ::= \star \mid x \mid \lambda x.P \mid \lambda !x.P \mid !V$$
$$P ::= \mathbf{0} \mid P \mid\mid P \mid a\langle x\rangle.P \mid a\langle !x\rangle.P \mid \overline{a}\langle V\rangle.P \mid (\nu a)P \mid VV$$

where

$$\begin{matrix}\lambda x.P \\ a(x).P\end{matrix} \longrightarrow \begin{matrix}x \text{ occurs once in } P \\ \text{at level } 0\end{matrix}$$

- Examples:

$a\langle x\rangle.x\star$	✓
$a\langle !x\rangle.(x\star \mid\mid !x\star)$	✓
$a\langle !x\rangle.\overline{a}\langle x\rangle.\overline{b}\langle x\rangle.\mathbf{0}$	✓
$a\langle !x\rangle.(b\langle y\rangle.\overline{c}\langle y\rangle.x\star \mid\mid \overline{a}\langle !x\rangle.\mathbf{0})$	✓
$a\langle x\rangle.(!x)\star$	$\lightning$

Linear Higher-Order π-Calculus : $\mathsf{LHO}\pi$

- Reduction:

$$\frac{}{\overline{a}\langle V\rangle.P \mid\mid a\langle x\rangle.Q \rightarrow_{\mathsf{L}} P \mid\mid Q[x/V]}$$

$$\frac{}{\overline{a}\langle !V\rangle.P \mid\mid a\langle !x\rangle.Q \rightarrow_{\mathsf{L}} P \mid\mid Q[x/V]}$$

$$\frac{}{(\lambda x.P)V \rightarrow_{\mathsf{L}} P[x/V]} \qquad \frac{}{(\lambda !x.P)!V \rightarrow_{\mathsf{L}} P[x/V]}$$

$$\frac{P \rightarrow_{\mathsf{L}} Q}{P \mid\mid R \rightarrow_{\mathsf{L}} Q \mid\mid R} \qquad \frac{P \rightarrow_{\mathsf{L}} Q}{(\nu a)P \rightarrow_{\mathsf{L}} (\nu a)Q}$$

$$\frac{P \equiv Q \quad Q \rightarrow_{\mathsf{L}} R \quad R \equiv S}{P \rightarrow_{\mathsf{L}} S}$$

Embedding $\mathsf{LHO}\pi$ Into $\mathsf{HO}\pi$

$$
\begin{aligned}
[\star]_{\mathsf{V}} &= \star \\
[\lambda x.P]_{\mathsf{V}} &= \lambda !x.[P]_{\mathsf{P}} \\
[\mathbf{0}]_{\mathsf{P}} &= \mathbf{0} \\
[x]_{\mathsf{P}} &= x \\
[P \mid\mid Q]_{\mathsf{P}} &= [P]_{\mathsf{P}} \mid\mid [Q]_{\mathsf{P}} \\
[a\langle x\rangle.P]_{\mathsf{P}} &= a\langle !x\rangle.[P]_{\mathsf{P}} \\
[\overline{a}\langle V\rangle.P]_{\mathsf{P}} &= \overline{a}\langle ![V]_{\mathsf{V}}\rangle.[P]_{\mathsf{P}} \\
[(\nu a)P]_{\mathsf{P}} &= (\nu a)[P]_{\mathsf{P}} \\
[VV]_{\mathsf{P}} &= [V]_{\mathsf{V}}![V]_{\mathsf{V}}
\end{aligned}
$$

Proposition (Simulation)

If $P \rightarrow_{\mathsf{P}} Q$, *then* $[P]_{\mathsf{P}} \rightarrow_{\mathsf{L}} [Q]_{\mathsf{P}}$

Soft Processes: SHOπ

- Generalizing $\Lambda_! \Longrightarrow \Lambda_S$ to processes.
- Linear processes with some additional constraints:

$$\begin{tikzcd}
\begin{array}{c}\lambda x.P \\ a(x).P\end{array} \arrow[r] & \begin{array}{c}x \text{ occurs once in } P \\ \text{at level } 0\end{array} \\
 & \begin{array}{c}x \text{ occurs once in } P \\ \text{at level } 1\end{array} \\
\begin{array}{c}\lambda !x.P \\ a(!x).P\end{array} \arrow[ur] \arrow[dr] & \\
 & \begin{array}{c}x \text{ occurs in } P \\ \text{at level } 0\end{array}
\end{tikzcd}$$

Soft Processes: Examples

$$a\langle x\rangle.x\star \qquad \checkmark$$

Soft Processes: Examples

$$a\langle x\rangle.x\star \qquad ✓$$

$$a\langle !x\rangle.(x\star \;||\; (!x)\star) \qquad \lightning$$

Soft Processes: Examples

$a\langle x\rangle.x\,\star$ ✓

$a\langle !x\rangle.(x\,\star \;||\; (!x)\star)$ $\lightning$

$a\langle !x\rangle.\overline{a}\langle x\rangle.\overline{b}\langle x\rangle.\mathbf{0}$ ✓

Soft Processes: Examples

$$a\langle x\rangle.x\star \qquad \checkmark$$

$$a\langle !x\rangle.(x\star \;||\; (!x)\star) \qquad \lightning$$

$$a\langle !x\rangle.\overline{a}\langle x\rangle.\overline{b}\langle x\rangle.\mathbf{0} \qquad \checkmark$$

$$a\langle !x\rangle.(b\langle y\rangle.\overline{c}\langle y\rangle.x \;||\; \overline{a}\langle !x\rangle) \qquad \lightning$$

Polytime Soundness

Theorem

There is a family of polynomials $\{p_n\}_n$ *such that for every process* P *and for every* m, *if* $P \rightarrow_{\mathsf{L}}^{m} Q$, *then* $m, |Q| \leq p_{\mathbb{B}(P)}(|P|)$.

Polytime Soundness

Theorem

There is a family of polynomials $\{p_n\}_n$ such that for every process P and for every m, if $P \rightarrow_{\mathsf{L}}^{m} Q$, then $m, |Q| \leq p_{\mathbb{B}(P)}(|P|)$.

- We can generalize polytime soundness by considering labelled semantics

$$P \overset{\overline{a}\langle R\rangle}{\rightarrow} Q \qquad\qquad P \overset{a\langle R\rangle}{\rightarrow} Q$$

Theorem

If $P = P_0 \overset{\alpha_1}{\rightarrow} \ldots \overset{\alpha_n}{\rightarrow} P_n$ and the input actions among $\alpha_1, \ldots, \alpha_n$ are $a_1\langle R_1\rangle, \ldots, a_m\langle R_m\rangle$, then

$$n, |P_i| \leq p_{\max\{\mathbb{B}(P), \mathbb{B}(R_i)\}}(|P| + \sum_{i=1}^{m} |R_i|)$$

Concurrent ICC: Challenges

- Does it make sense to measure time in terms of the number of *interleaving* reduction steps?

Concurrent ICC: Challenges

- Does it make sense to measure time in terms of the number of *interleaving* reduction steps?
- The answer is negative: one should rather count “how much computation” stimuli from the environment (i.e., messages) can possibly *trigger*.

Concurrent ICC: Challenges

- Does it make sense to measure time in terms of the number of *interleaving* reduction steps?
- The answer is negative: one should rather count "how much computation" stimuli from the environment (i.e., messages) can possibly *trigger*.
- Linking events to the "complexity" triggered by them is impossible in interleaving semantics, and one is then forced to go towards true concurrency [DLHMV12].
- A recent contribution [DemangeonYoshida18] is the first one seriously going in this direction.

Concurrent ICC: Challenges

- Does it make sense to measure time in terms of the number of *interleaving* reduction steps?
- The answer is negative: one should rather count "how much computation" stimuli from the environment (i.e., messages) can possibly *trigger*.
- Linking events to the "complexity" triggered by them is impossible in interleaving semantics, and one is then forced to go towards true concurrency [DLHMV12].
- A recent contribution [DemangeonYoshida18] is the first one seriously going in this direction.
- Another excellent research topic!

Thank you!

Questions?