

A PolyTime Functional Language from Light Linear Logic*

Patrick Baillot[!], Marco Gaboardi[†] and Virgile Mogbil[§]

[!]École Normale Supérieure de Lyon

[†]Università di Bologna

[§]Université Paris 13

PICS "Logique linéaire et applications"
CONCERTO Controllo e certificazione dell'uso delle risorse

June 9, 2010, Torino

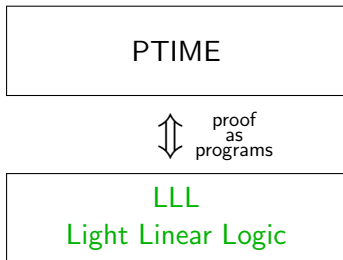
(presented at ESOP10, March 2010, Paphos, Cyprus)

*Partially supported by project ANR-08-BLANC-0211-01 "ComplICE"

A PolyTime Functional Language from Light Linear Logic*

Aim

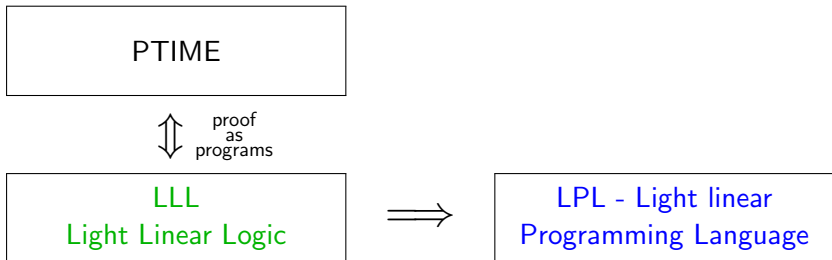
inspired by a **linear logic based characterization (LLL)** of the PTIME class.



A PolyTime Functional Language from Light Linear Logic*

Aim

Design a **concrete functional language** for polynomial time computations inspired by a **linear logic based characterization (LLL)** of the PTIME class.



A PolyTime Functional Language

from Light Linear Logic*

Aim

Design a **concrete functional language** for polynomial time computations inspired by a **linear logic based characterization (LLL)** of the PTIME class.

Outline

- Introduction and background: Light Linear Logic
- Light linear Programming Language (**LPL**) and its Type System
- Main results and proof ideas

The general setting

- **Implicit Computational Complexity (ICC)** aims at characterizing **complexity properties** by restrictions on programming languages constructions.

General goals:

- ▶ characterize complexity classes of functions, e.g. PTIME, PSPACE
- ▶ design methods to statically guarantee complexity properties of programs

The general setting

- **Implicit Computational Complexity (ICC)** aims at characterizing **complexity properties** by restrictions on programming languages constructions.

General goals:

- ▶ characterize complexity classes of functions, e.g. PTIME, PSPACE
 - ▶ design methods to statically guarantee complexity properties of programs
- It generally borrows techniques from **Mathematical Logic** :
 - ▶ Recursion Theory
 - ▶ Structural Proof Theory
 - ▶ Model Theory

Functional characterization of the class PTIME

Higher-Order Calculi:

- ① **light linear logic** approach (**LLL** [Girard 98], **SLL** [Lafont 04]): provide term languages (**λ -light** [Terui 01], **λ -soft** [Baillot-Mogbil 04]) or type systems as criteria (**DLAL** [Baillot-Terui 09], **STA** [Gaboardi-Ronchi della Rocha 07]),

Functional characterization of the class PTIME

Higher-Order Calculi:

- 1 **light linear logic** approach (**LLL** [Girard 98], **SLL** [Lafont 04]): provide term languages (**λ -light** [Terui 01], **λ -soft** [Baillot-Mogbil 04]) or type systems as criteria (**DLAL** [Baillot-Terui 09], **STA** [Gaboardi-Ronchi della Rocha 07]),

PROS: Proof-as-program paradigm \Rightarrow higher-order and polymorphism. CONS: Algorithms represented are limited: not so user-friendly.

Functional characterization of the class PTIME

Higher-Order Calculi:

- 1 **light linear logic** approach (**LLL** [Girard 98], **SLL** [Lafont 04]): provide term languages (**λ -light** [Terui 01], **λ -soft** [Baillot-Mogbil 04]) or type systems as criteria (**DLAL** [Baillot-Terui 09], **STA** [Gaboardi-Ronchi della Rocha 07]),
- 2 **ramification** [Bellantoni-Cook 92, Leivant 91] approach and/or **inspired from system T** (safe recursion with higher-order types **SLR** [Hofmann 00, Bellantoni-Niggli-Schwichtenberg 00], **linear HO** [Dal Lago-Martini-Roversi 03, Dal Lago 09]), non-size-increasing **LFPL** [Hofmann 99]).

Functional characterization of the class PTIME

Higher-Order Calculi:

- 1 **light linear logic** approach (**LLL** [Girard 98], **SLL** [Lafont 04]): provide term languages (**λ -light** [Terui 01], **λ -soft** [Baillot-Mogbil 04]) or type systems as criteria (**DLAL** [Baillot-Terui 09], **STA** [Gaboardi-Ronchi della Rocha 07]),
- 2 **ramification** [Bellantoni-Cook 92, Leivant 91] approach and/or **inspired from system T** (safe recursion with higher-order types **SLR** [Hofmann 00, Bellantoni-Niggli-Schwichtenberg 00], **linear HO** [Dal Lago-Martini-Roversi 03, Dal Lago 09]), non-size-increasing **LFPL** [Hofmann 99]).

CONS: higher-order is quite constrained (linearity), AND/OR: from system T \Rightarrow not easy to program, far from ordinary functional languages.

Functional characterization of the class PTIME

1st Order Calculi:

- ① **Quasi-interpretations** [Marion-Moyen 00, Bonfante-Marion-Moyen 01, Amadio 05] and **sup-interpretations** [Marion-Péchox 08,09] provide functional languages with recursion and pattern matching.

Functional characterization of the class PTIME

1st Order Calculi:

- 1 **Quasi-interpretations** [Marion-Moyen 00, Bonfante-Marion-Moyen 01, Amadio 05] and **sup-interpretations** [Marion-Péchox 08,09] provide functional languages with recursion and pattern matching.

PROS: expressivity on the considered programs, thanks to a combined criterion: *termination condition + size bound*.

CONS: no higher-order, type checking easier than checking of quasi/sup-interpretations.

Bring together the PROS ?

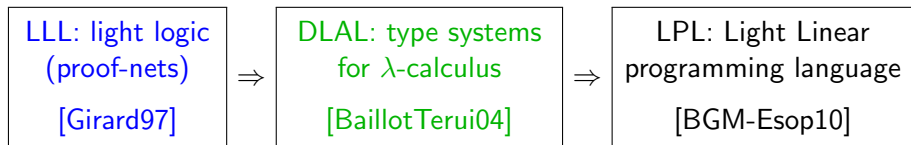
Refined goals:

- using LLL to bring together **higher-order and recursion** style programming, with **pattern-matching**, in a functional language with guaranteed PTIME bounds

Bring together the PROS ?

Refined goals:

- using LLL to bring together **higher-order and recursion** style programming, with **pattern-matching**, in a functional language with guaranteed PTIME bounds



Approach:

- we are not aiming at an encoding of a language into LLL or DLAL
- we would **follow** LLL architecture and principles
- we would **combine** syntactic and typing conditions

A difficulty in dealing with λ -calculus and recursion

We can easily combine apparently harmless terms to obtain **exponential blow up**.

Example 1

Take the recursive definition of `mul` on numerals \underline{n} and consider:

$$\lambda x.x(\lambda y.\text{mul } \underline{2} y)\underline{1}$$

It is apparently harmless!

But for each Church numeral $\underline{c}_n = \lambda s.\lambda z.s^n z$ it returns the numeral $\underline{2}^n$.

A **tight control on both** recursive calls and β -reduction steps is needed.

A difficulty in dealing with λ -calculus and recursion

We can easily combine apparently harmless terms to obtain **exponential blow up**.

Example 2

Take `ListOf2` that given numeral n returns a list of 2 of length n and `foldr` defined as usual. The following term is exponential in its argument.

$$\lambda x. \text{foldr mul 1 (ListOf2 x)}$$

Types can be used to prevent this.

Light Linear Logic (LLL)

Types we use here:

$$A ::= \mathbb{D} \mid A \multimap B \mid !A \multimap B \mid \S A$$
$$\mathbb{D} ::= \mathbb{N} \mid \mathbb{W} \mid \mathbb{L} \mid \mathbb{T} \quad (\text{base data types})$$

The logical rules (for modalities ! and \S) imply that:

- a non-linear argument (of a $!A \multimap B$) has at most one free variable
- iteration:

$$\frac{\text{step} : A \multimap A \quad \text{base} : \S A}{\mathbb{N} \multimap \S A}$$

an *iteration* cannot be used as the *step* of another iteration

type	time bound	size bound on result
$\mathbb{N} \multimap \mathbb{N}$	$O(n^2)$	$O(n)$
$\mathbb{N} \multimap \S \mathbb{N}$	$O(n^4)$	$O(n^4)$
$\mathbb{N} \multimap \S^d \mathbb{N}$	$O(n^{2^{d+1}})$	$O(n^{2^{d+1}})$

LLL proof nets

- Every proof-net Π has a maximal level d (**depth**) and is **stratified** into levels $0, 1, \dots, d$
 - ▶ level i does not depend from levels $j > i$
 - ▶ potential complexity: contraction nodes, i.e. a contraction node at level i can duplicate objects at levels $j > i$
- The evaluation can be done **level-by-level**, in successive rounds: $0, 1, \dots, d$.
- After the round at each level, the size increases quadratically.
- We get an overall bound $O(|\Pi|^{2^d})$ both for **size** and **computation time**.

Note: in LLL, **data types have a fixed depth**. This means that the exponent d depends just on the program part, hence we can work in **polynomial time**.

LPL: a small functional programming language

The Language:

M, N	$::=$	$x \mid c \ t_1 \cdots t_n \mid X \mid F \ t_1 \cdots t_n \mid \lambda x.M \mid MM$	terms
v	$::=$	$c \ v_1 \cdots v_n$	values
t	$::=$	$X \mid c \ t_1 \cdots t_n$	patterns
d_F	$::=$	$F \ t_1 \cdots t_n = N$	function def.
p	$::=$	$M \mid \text{LetRec } d_F, \dots, d_F \text{ in } p$	programs

- There is no mutually defined functions
- the scope of LetRec is static and global.
- Patterns are *linear*: X occurs at most once in $c \ t_1 \ \dots \ t_n$
- We consider a reduction mechanism consisting in the usual β -reduction and a **LetRec_F matching reduction**.

Examples

- Addition:

```
LetRec Add (x + 1) y = (Add x y) + 1 ,  
      Add 0 y = y  
      in Add (Add 3 2) 4
```

- Map:

```
LetRec Map f (x : xs) = (fx) : (Map f xs) ,  
      Map f nil = nil  
      in Map (+1) (1 : 1 : 0 : 1)
```

- Append on lists:

```
LetRec Append (x : xs) ys = x : (Append xs ys) ,  
      Append nil ys = ys  
      in Append (Append (1 : nil) (0 : 1 : nil)) (1 : nil)
```

The combined criterion

The difficulty:

A reduction **mixing** of β -reduction and rewriting (LetRec_F).

The solution:

Combined Criterion

Syntactic Termination
Condition

Light Linear Typing
Condition

Controlling the number
of recursive steps for
individual recursive calls

Controlling β -steps
and the nesting
of recursions

The syntactic criterion

The *syntactic termination criterion* checks for every definition:

$$F \ t_1 \dots t_n = M$$

that

each recursive call $F \ t_1^i \dots t_n^i$ in the term M

is done on a distinct strict sub-pattern of $t_1 \dots t_n$.

$$\text{Div } (s \ (s \ X)) = s \ (\text{Div } X)$$

$$\text{Div } (s \ 0) = 0$$

$$\text{Div } 0 = 0$$

$$\text{Min } (s \ X) \ (s \ Y) = s \ (\text{Min } X \ Y)$$

$$\text{Min } (s \ X) \ 0 = 0$$

$$\text{Min } 0 \ (s \ Y) = 0$$

The syntactic criterion

The *syntactic termination criterion* checks for every definition:

$$F \ t_1 \dots t_n = M$$

that

each recursive call $F \ t_1^i \dots t_n^i$ in the term M

is done on a distinct strict sub-pattern of $t_1 \dots t_n$.

But also as:

$\text{Foldr } F \ Z \ (X : XS) = F \ X \ (\text{Foldr } F \ Z \ XS)$, $X : XS$ is a recurrence arg.

$\text{Foldr } F \ Z \ \text{nil} = Z$

$\text{Tadd} \ (\text{node } X \ L \ R) \ (\text{node } X' \ L' \ R') = \text{node } (X + X') \ (\text{Tadd } L \ L') \ (\text{Tadd } R \ R')$

$\text{Tadd } \epsilon \ X = X$

$\text{Tadd } X \ \epsilon = X$

Typing condition

The LPL types are:

$$\begin{aligned} A &::= \mathbb{D} \mid A \multimap B \mid !A \multimap B \mid \S A \\ \mathbb{D} &::= \mathbb{N} \mid \mathbb{W} \mid \mathbb{L} \mid \dots \end{aligned} \quad (\text{base data types})$$

Typing rules for constructors and function symbols:

$$\overline{\vdash c : \mathcal{T}(c)} \quad \overline{\vdash F : \mathcal{T}(F)}$$

- constructors: $\mathcal{T}(c) = \mathbb{D}_1 \multimap \dots \multimap \mathbb{D}_n \multimap \mathbb{D}_{n+1}$
- functions: where $!^i \S^j$ is denoted by \dagger^{i+j} ,

$$\mathcal{T}(F) = \dagger^{i_1} A_1 \multimap \dots \multimap \dagger^{i_n} A_n \multimap \S^j A$$

- ▶ $j \geq 1$
- ▶ for every recurrence argument k we have $i_k = 0$
- ▶ for every other argument r we have $i_r \geq 1$

Typing rules

λ -calculus part: DLAL (extended to pattern variables)

Recursive definitions and programs:

$$\frac{\Theta_1; \Theta_2 \vdash F \vec{t} : C \quad \Theta_1; \Theta_2 \vdash N : C}{\triangleright (F \vec{t} = N) : C} (\star)(D)$$

$$\frac{\Gamma; \Delta \vdash p : C \quad \triangleright d_F : C \cdots \triangleright d_F : C}{\text{LetRec } d_F, \dots, d_F \text{ in } p} (R)$$

- in (D) : Θ_1, Θ_2 only contain pattern variables
- (\star) : condition on the recurrence arguments should be satisfied

Examples of types

- some function types:

$$\text{Add } (x + 1) \ y = (\text{Add } x \ y) + 1 ,$$

$$\text{Add } 0 \ y = y$$

$$\vdash \text{Add} : \mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}$$

$$\text{Map } f \ (x : xs) = (fx) : (\text{Map } f \ xs) ,$$

$$\text{Map } f \ \text{nil} = \text{nil}$$

$$\vdash \text{Map} : !(\mathbb{N} \multimap \mathbb{N}) \multimap \mathbb{L} \multimap \mathbb{L}$$

$$\text{Foldr } f \ z \ (x : xs) = f \ (\text{coer } x) \ (\text{Foldr } f \ z \ xs) ,$$

$$\text{Foldr } f \ z \ \text{nil} = z$$

$$\vdash \text{Foldr} : !(\mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}) \multimap \mathbb{N} \multimap \mathbb{L} \multimap \mathbb{N}$$

- example of program type:

LetRec $d_{\text{Add}}, d_{\text{Foldr}}$ in $\text{Foldr } (\lambda x. \lambda y. \text{Add } x \ y) \ 0 \ (0 : 1 : \dots : n) : \mathbb{N}$

From typed programs to executable terms

Source language

typed program

→

intermediate language

executable term

depth structure is made explicit
analogous to proof-nets

$\lambda x.f(fx) : \S(\alpha \multimap \alpha)$

let f be !f' in $\S(\lambda x.f'(f'x)) : \S(\alpha \multimap \alpha)$

The normalization proof idea: Adapting the proof for LLL:

- replace proof-net depth by a new invariant: *potential depth* d
- reduction strategy: *stratified reduction*, i.e. depth $0, 1, \dots, d$
- at each depth i , we use *2 measures*:

$|s|_i$ to bound β steps

and

$SA_i^F(M)$ to bound LetRec_F steps

Results

Theorem (Soundness)

Consider a program $p = \text{LetRec } d_{F_1}, \dots, d_{F_n} \text{ in } M$ satisfying:

- the *syntactic criterion*
- the *typing condition*, with potential depth d ,

then there *exists a polynomial P* such that p can be reduced in a number of steps bounded by $P(|p|)$. The polynomial P only depends on the potential depth d .

In particular, if $p : \mathbb{W} \multimap \S^k \mathbb{W}$ then p represents a *PTIME function*.

Theorem (Completeness)

For any *PTIME function* $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, there exists an integer k and an *LPL program* of type $\mathbb{W} \multimap \S^k \mathbb{W}$, representing f .

Conclusion and perspectives

- We have defined a **higher-order functional language** with recursively defined functions and with a **criterion for Ptime**.
- This is a setting in which the techniques coming from different approaches (quasi-interpretations, non-size-increasing, linear logic) could be combined.
- The bound induced by typing is rough, but it could be **efficiently** inferred.
- Some improvements:
 - ▶ both the syntactic and typing criterion can be **relaxed**
 - ▶ **integer** and **symbolic** constraints can be added to achieve **finer bounds**
- Perspectives for functional languages:
 - ▶ space resource analysis
 - ▶ resource control techniques