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 linaire 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.

PTIME

$\Updownarrow$ proof as programs

LLL
Light Linear Logic

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.

PTIME

$\Updownarrow$ proof as programs

LLL
Light Linear Logic

$\Longrightarrow$

LPL - Light linear
Programming Language

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-Niggl-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-Niggl-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-Péchoux 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-Péchoux 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

LLL: light logic (proof-nets) [Girard97]	$\Rightarrow$	DLAL: type systems for λ-calculus [BaillotTerui04]	$\Rightarrow$	LPL: Light Linear programming language [BGM-Esop10]

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 $\underline{\mathtt{n}}$ and consider:

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

It is apparently harmless!

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

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 $\underline{\mathtt{n}}$ returns a list of 2 of length *n* and `foldr` defined as usual. The following term is exponential in its argument.

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

Types can be used to prevent this.

Light Linear Logic (LLL)

Types we use here:

$$\begin{array}{rcll} A & ::= & \mathbb{D} \mid A \multimap B \mid !A \multimap B \mid \S A & \\ \mathbb{D} & ::= & \mathbb{N} \mid \mathbb{W} \mid \mathbb{L} \mid \mathbb{T} & \qquad (\textit{base data types}) \end{array}$$

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

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

$$\frac{\textit{step} : A \multimap A \qquad \textit{base} : \S A}{\mathbb{N} \multimap \S A}$$

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 $0, 1, \ldots, d$
  - 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: $0, 1, \ldots, d$.
- 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:

$$
\begin{array}{rcll}
\mathtt{M}, \mathtt{N} & ::= & \mathtt{x} \mid \mathtt{c}\ \mathtt{t}_1 \cdots \mathtt{t}_n \mid \mathtt{X} \mid \mathtt{F}\ \mathtt{t}_1 \cdots \mathtt{t}_n \mid \lambda \mathtt{x}.\mathtt{M} \mid \mathtt{M}\mathtt{M} & \text{terms} \\
\mathtt{v} & ::= & \mathtt{c}\ \mathtt{v}_1 \cdots \mathtt{v}_n & \text{values} \\
\mathtt{t} & ::= & \mathtt{X} \mid \mathtt{c}\ \mathtt{t}_1 \ldots \mathtt{t}_n & \text{patterns} \\
\mathtt{d}_\mathtt{F} & ::= & \mathtt{F}\ \mathtt{t}_1 \ldots \mathtt{t}_n = \mathtt{N} & \text{function def.} \\
\mathtt{p} & ::= & \mathtt{M} \mid \mathtt{LetRec}\ \mathtt{d}_\mathtt{F}, \ldots, \mathtt{d}_\mathtt{F}\ \mathtt{in}\ \mathtt{p} & \text{programs}
\end{array}
$$

- There is no mutually defined functions
- the scope of `LetRec` is static and global.
- Patterns are *linear*: $\mathtt{X}$ occurs at most once in $\mathtt{c}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n$
- We consider a reduction mechanism consisting in the usual β-reduction and a $\mathtt{LetRec}_\mathtt{F}$ matching reduction.

Examples

- Addition:

$$\begin{aligned}\mathtt{LetRec}\ \mathtt{Add}\ (\mathtt{x}+1)\ \mathtt{y} &= (\mathtt{Add}\ \mathtt{x}\ \mathtt{y})+1\ ,\\ \mathtt{Add}\ 0\ \mathtt{y} &= \mathtt{y}\\ &\mathtt{in}\ \mathtt{Add}\ (\mathtt{Add}\ 3\ 2)\ 4\end{aligned}$$

- Map:

$$\begin{aligned}\mathtt{LetRec}\ \mathtt{Map}\ \mathtt{f}\ (\mathtt{x}:\mathtt{xs}) &= (\mathtt{fx}):(\mathtt{Map}\ \mathtt{f}\ \mathtt{xs})\ ,\\ \mathtt{Map}\ \mathtt{f}\ \mathtt{nil} &= \mathtt{nil}\\ &\mathtt{in}\ \mathtt{Map}\ (+1)\ (1:1:0:1)\end{aligned}$$

- Append on lists:

$$\begin{aligned}\mathtt{LetRec}\ \mathtt{Append}\ (\mathtt{x}:\mathtt{xs})\ \mathtt{ys} &= \mathtt{x}:(\mathtt{Append}\ \mathtt{xs}\ \mathtt{ys})\ ,\\ \mathtt{Append}\ \mathtt{nil}\ \mathtt{ys} &= \mathtt{ys}\\ &\mathtt{in}\ \mathtt{Append}\ (\mathtt{Append}\ (1:\mathtt{nil})\ (0:1:\mathtt{nil}))\ (1:\mathtt{nil})\end{aligned}$$

The combined criterion

The difficulty:
A reduction mixing of β-reduction and rewriting ($\texttt{LetRec}_{\texttt{F}}$).

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:

$$\mathtt{F}\ \mathtt{t}_1 \ldots \mathtt{t}_n = \mathtt{M}$$

that

each recursive call $\mathtt{F}\ \mathtt{t}^i_1 \ldots \mathtt{t}^i_n$ in the term $\mathtt{M}$

is done on a distinct strict sub-pattern of $\mathtt{t}_1 \ldots \mathtt{t}_n$.

$\mathtt{Div}\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{X})) = \mathtt{s}\ (\mathtt{Div}\ \mathtt{X})$
$\mathtt{Div}\ (\mathtt{s}\ 0) = 0$
$\mathtt{Div}\ 0 = 0$

$\mathtt{Min}\ (\mathtt{s}\ \mathtt{X})\ (\mathtt{s}\ \mathtt{Y}) = \mathtt{s}\ (\mathtt{Min}\ \mathtt{X}\ \mathtt{Y})$
$\mathtt{Min}\ (\mathtt{s}\ \mathtt{X})\ 0 = 0$
$\mathtt{Min}\ 0\ (\mathtt{s}\ \mathtt{Y}) = 0$

The syntactic criterion

The *syntactic termination criterion* checks for every definition:

$$\mathtt{F}\ \mathtt{t}_1 \ldots \mathtt{t}_\mathtt{n} = \mathtt{M}$$

that

$$\text{each recursive call } \mathtt{F}\ \mathtt{t}^\mathtt{i}_1 \ldots \mathtt{t}^\mathtt{i}_\mathtt{n} \text{ in the term } \mathtt{M}$$

is done on a distinct strict sub-pattern of $\mathtt{t}_1 \ldots \mathtt{t}_\mathtt{n}$.

But also as:

$\mathtt{Foldr}\ \mathtt{F}\ \mathtt{Z}\ (\mathtt{X} : \mathtt{XS}) = \mathtt{F}\ \ \mathtt{X}\ (\mathtt{Foldr}\ \mathtt{F}\ \mathtt{Z}\ \mathtt{XS})$, $\quad \mathtt{X} : \mathtt{XS}$ *is a recurrence arg.*
$\mathtt{Foldr}\ \mathtt{F}\ \mathtt{Z}\ \mathtt{nil} = \mathtt{Z}$

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

Typing condition

The LPL types are:

$$\begin{array}{lcll} A & ::= & \mathbb{D} \mid A \multimap B \mid !A \multimap B \mid \S A & \\ \mathbb{D} & ::= & \mathbb{N} \mid \mathbb{W} \mid \mathbb{L} \mid \ldots & \textit{(base data types)} \end{array}$$

Typing rules for constructors and function symbols:

$$\frac{}{\vdash \mathtt{c} : \mathcal{T}(\mathtt{c})} \qquad \frac{}{\vdash \mathtt{F} : \mathcal{T}(\mathtt{F})}$$

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

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

  - $j \geq 1$
  - 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:

$$\frac{\Theta_1; \Theta_2 \ \vdash \ \mathtt{F}\ \overrightarrow{t} : C \qquad \Theta_1; \Theta_2 \ \vdash \ \mathtt{N} : C}{\triangleright\, (\mathtt{F}\ \overrightarrow{t}\ = \ \mathtt{N}) : C}\ (\star)(D)$$

$$\frac{\Gamma; \Delta \ \vdash \ \mathtt{p} : C \qquad \triangleright\, \mathtt{d_F} : C \ \cdots \ \triangleright\, \mathtt{d_F} : C}{\mathtt{LetRec}\ \mathtt{d_F}, \ldots, \mathtt{d_F}\ \mathtt{in}\ \mathtt{p}}\ (R)$$

- 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:

$$
\begin{aligned}
\texttt{Add}\ (x+1)\ y &= (\texttt{Add}\ x\ y)+1\ , \\
\texttt{Add}\ 0\ y &= y \\
\vdash \texttt{Add} &: \mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{Map}\ f\ (x : xs) &= (f x) : (\texttt{Map}\ f\ xs)\ , \\
\texttt{Map}\ f\ \texttt{nil} &= \texttt{nil} \\
\vdash \texttt{Map} &: !(\mathbb{N} \multimap \mathbb{N}) \multimap \mathbb{L} \multimap \S\mathbb{L}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{Foldr}\ f\ z\ (x : xs) &= f\ (\texttt{coer}\ x)\ (\texttt{Foldr}\ f\ z\ xs)\ , \\
\texttt{Foldr}\ f\ z\ \texttt{nil} &= z \\
\vdash \texttt{Foldr} &: !(\mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}) \multimap \S\S\mathbb{N} \multimap \mathbb{L} \multimap \S\S\mathbb{N}
\end{aligned}
$$

- example of program type:

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

From typed programs to executable terms

Source language		**intermediate language**
typed program	$\longrightarrow$	executable term
		depth structure is made explicit analogous to proof-nets
$\lambda \mathtt{x}.\mathtt{f}(\mathtt{f}\mathtt{x}) : \S(\alpha \multimap \alpha)$		$\mathtt{let\ f\ be\ !f'\ in\ } \S(\lambda \mathtt{x}.\mathtt{f}'(\mathtt{f}'\mathtt{x})) : \S(\alpha \multimap \alpha)$

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, . . . , *d*
- at each depth *i*, we use *2 measures*:

$|s|_i$ to bound β steps and $\mathrm{SA}_{\mathtt{i}}^{\mathtt{F}}(\mathtt{M})$ to bound $\mathtt{LetRec}_{\mathtt{F}}$ steps

Results

Theorem (Soundness)

Consider a program $\mathrm{p} = \texttt{LetRec}\ \mathtt{d}_{\mathtt{F}_1}, \ldots \mathtt{d}_{\mathtt{F}_n}\ \texttt{in}\ \mathtt{M}$ *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 $P(|\mathrm{p}|)$. The polynomial P only depends on the potential depth d.*

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

Theorem (Completeness)

For any PTIME function $f : \{0,1\}^\star \to \{0,1\}^\star$, 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