Linear dependent types and intensional expressivity

Ugo Dal Lago and Marco Gaboardi

Dipartimento di Scienze dell'Informazione - Università di Bologna

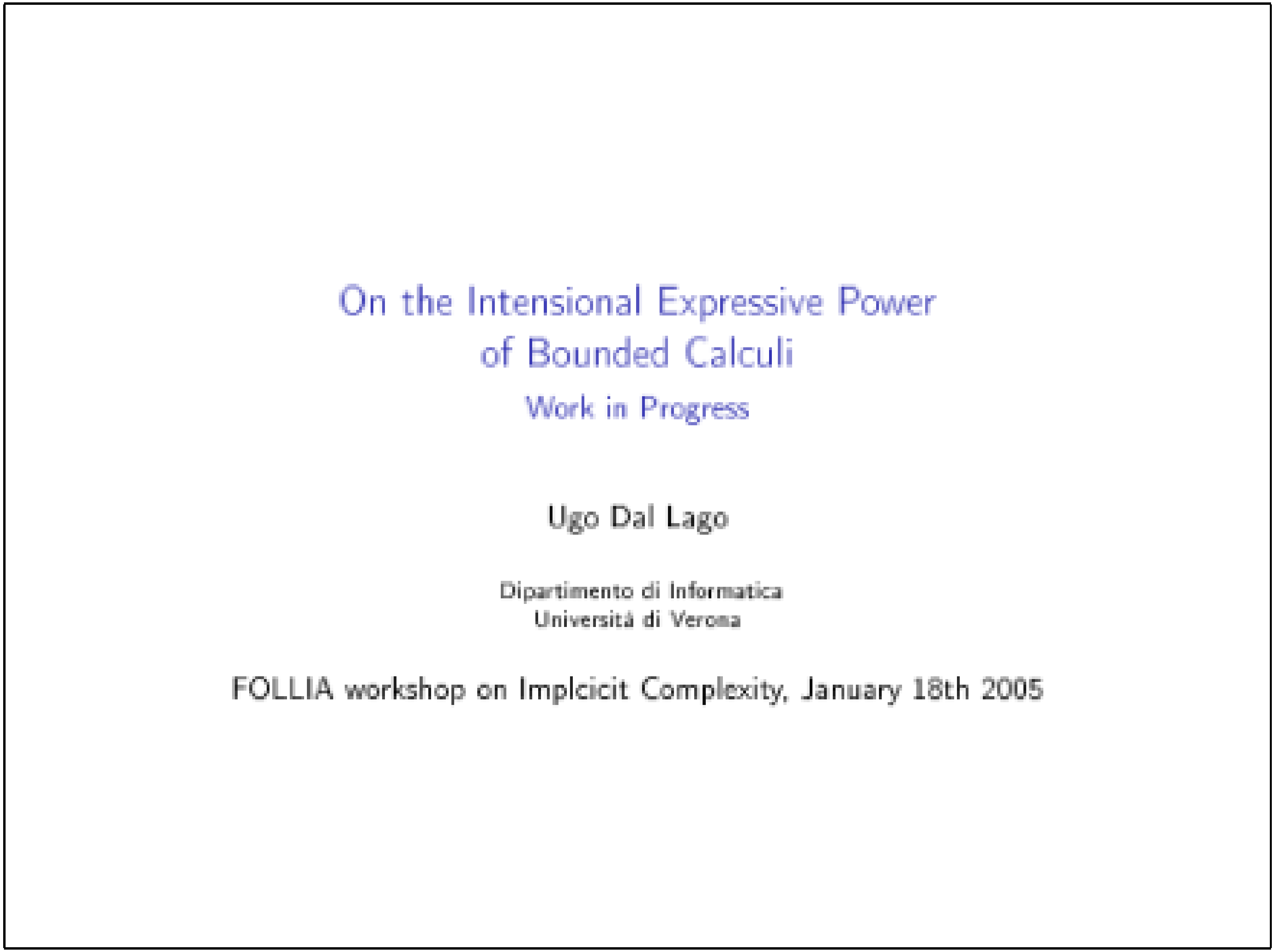
Aim of this presentation

I present a work in (very) progress that aims at provide a general framework for implicit computational complexity.

What you need to retain:

"Intensional completeness is important and we can treat it in non trivial ways"

Four years ago...



Classical Implicit Computational Complexity

- The usual scenario:
  - You pick up a complexity class $\mathcal{C}$, e.g. PTIME, LOGSPACE, etc.
  - Starting from a programming language or logical system $\mathcal{P}$, you isolate a subclass $\mathcal{P}^*$ of $\mathcal{P}$.
  - Then, you prove:

    Soundness: that every program in $\mathcal{P}^*$ is computable in time (or space) bounded by a function in $\mathcal{C}$.

    Completeness: every function computable in time (or space) bounded by a function in $\mathcal{C}$ is representable in $\mathcal{P}^*$

- Here we have a mismatch!

  Soundness is proved intensionally but completeness is proved extensionally.

Towards Intensionality

- We can replace the usual statement:

  > Completeness: Every function computable in time (or space) bounded by a function in $\mathcal{C}$ is shown to be representable in $\mathcal{P}^*$.

  with the following one

  > Intensional Completeness: the subclass $\mathcal{P}^*$ contains every $\mathcal{P}$ program computable in time (or space) bounded by a function in $\mathcal{C}$.

- Intensional completeness is far more interesting from a programming perspective!
- Unfortunately, if $\mathcal{P}^*$ is sound and intensionally complete, then $\mathcal{P}^*$ is not recursively enumerable, provided $\mathcal{P}^*$ and $\mathcal{C}$ are nontrivial.
- What should we do?

A paradigm shift...

- The proof assistant community has already considered a similar question:

  "how to deal with interesting intrinsically non decidable properties?"

- The Interactive Proof Assistants solution is to consider strong logical systems (e.g. CiC) where such properties could be described, loosing in this way full automatization.

- We think that a similar approach is needed here. By a slogan:

  "We need to consider seriously the development of intensional complete systems, leaving the decidability to an external task."

A first source of inspiration: Bounded Recursion on Notation

- In his seminal work, Cobham has introduced Bounded Recursion on Notation $\mathtt{BRN}$ as the functional counterpart of the class of function computable by a Deterministic Turing Machine in Polynomial time PTIME.
- Using a Recursion on Notation scheme, we can also define a language for the Primitive Recursive Functions. Let us name this system $\mathtt{PRN}$.
- Clearly, we have:

$$\mathtt{BRN} \subseteq \mathtt{PRN}$$

- In fact, we have something more. Bounded Recursion on Notation is intensionally hereditary polytime complete with respect to $\mathtt{PRN}$ programs, i.e. if a $\mathtt{PRN}$ program $\mathcal{P}$ and all its parts $\mathcal{P}_i$ are polytime, then $\mathcal{P} \in \mathtt{BRN}$.

A second source of inspiration: Bounded Linear Logic

Resource Polynomials:

$$p ::= \sum_{j \leq m} \prod_{i \leq k} \binom{x_{i,j}}{n_{i,j}}$$

Formulae:

$$\sigma, \tau ::= \alpha(\vec{p}) \mid \sigma \otimes \tau \mid \sigma \multimap \tau \mid \forall \alpha.\sigma \mid !_{x<p}\sigma$$

Rules:

$$\frac{\sigma \leq \sigma'}{\sigma \vdash \sigma'}\ (ax) \qquad \frac{\Gamma \vdash \tau}{\Gamma, !_{x<w}\sigma \vdash \tau}\ (w) \qquad \frac{\Gamma, \sigma[x := 0] \vdash \tau}{\Gamma, !_{x<1+w}\sigma \vdash \tau}\ (d)$$

$$\frac{\Gamma, !_{x<p}\sigma, !_{y<q}\sigma[x := p+y] \vdash \tau \quad p+y \text{ free for } x \text{ in } \sigma}{\Gamma, !_{x<p+q+w}\sigma \vdash \tau}\ (c)$$

$$\frac{!_{z<q_1(x)}\sigma_1[y := v_1(x) + z], \ldots, !_{z<q_n(x)}\sigma_n[y := v_n(x) + z] \vdash \tau}{!_{y<v_i(p)+w_1}\sigma_1, \ldots, !_{y<v_n(p)+w_n}\sigma_n \vdash !_{x<p}\tau}\ (p)$$

Dependent ML - 1

- In his phd thesis, Hongwei Xi has proposed a language, named Dependent ML (DML), obtained by extending the ML language and type system by means of a limited form of dependent types.
- The goal was to achieve a system where “it is possible to specify and infer precise type information facilitating program error detection and compiler optimization.”
- Concretely, dependent types appear in DML types and terms in the form of type index objects, built starting by a constraint index language $\mathcal{L}$, and in the form of universal and existential quantification over type index variables.
- The constraint index language $\mathcal{L}$ can be choose arbitrarily. E.g. linear inequalities over integers, boolean constraints, finite sets, etc.
- In this way type checking is reduced to constraint satisfaction in $\mathcal{L}$.

Dependent ML - An example

```
datatype 'a list = nil | cons of 'a * 'a list
typeref 'a list of nat with (* indexing the datatype 'a list with nat *)
    nil <| 'a list(0)
  | cons <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)

fun('a)
    append(nil, ys) = ys
  | append(cons(x, xs), ys) = cons(x, append(xs, ys))
where append <| {m:nat}{n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
```

$$
\begin{array}{l}
\textbf{fix}\ \mathit{append} : \Pi m : \mathit{nat}.\Pi n : \mathit{nat}.\mathit{intlist}(m) * \mathit{intlist}(n) \rightarrow \mathit{intlist}(m+n).\\
\quad \lambda m : \mathit{nat}.\lambda n : \mathit{nat}.\textbf{lam}\ l : \mathit{intlist}(m) * \mathit{intlist}(n).\\
\qquad \textbf{case}\ l\ \textbf{of}\\
\qquad\quad \langle \mathit{nil}, \mathit{ys}\rangle \Rightarrow \mathit{ys}\\
\qquad\quad \langle \mathit{cons}[a](\langle x, \mathit{xs}\rangle), \mathit{ys}\rangle \Rightarrow \mathit{cons}[a+n](\langle x, \mathit{append}[a][n](\langle \mathit{xs}, \mathit{ys}\rangle)\rangle)
\end{array}
$$

```
fun append[0][n](nil, ys) = ys
  | append[a+1][n](cons[a](x, xs), ys) = cons[a+n](x, append[a][n](xs, ys))
where append <| {m:nat}{n:nat} intlist(m) * intlist(n) -> intlist(m+n)
```

Dependent ML - 2

- Dependent types in DML appears in a very weak form, nevertheless they add an impressive reasoning power.
- In particular, the indexing terms can be used to statically capture several run time information about the program execution.
- The obtained information can be used to check whether the program satisfies certain properties or not.
- Conversely, by imposing a priori constraints on the shape of indexing terms only programs with an intended behaviour can be allowed.
- Unfortunately, DML seems not sufficient to reason about the implicit complexity of higher order programs.

Our scenario: combining Linearity and Dependent Types

Step 1: Take a simple (Typed) Turing Complete functional programming language, e.g. PCF or better for the moment a fixpoint free PCF.

Step 2: Dissect it through the usual linear logic decomposition.

Step 3: Decorate the type derivations by means of index terms, built by a particular constraint index language, representing information about program time bound and computed values.

Step 4: Check whether the constraints generated at the previous step can be satisfied or not.

Index language

$$
\begin{array}{lcll}
\gamma & ::= & o \mid \{a : \gamma \mid \mathrm{I}\} \mid \gamma + \gamma \mid \gamma \times \gamma & \text{index sort} \\
\mathcal{S} & ::= & \emptyset \mid \mathcal{S}, \mathtt{c}^n : \gamma_1 \times \cdots \times \gamma_n \to \gamma & \text{index signatures} \\
\mathrm{I} & ::= & a \mid 0 \mid \mathrm{I} + 1 \mid \mathrm{I}_1 \doteq \mathrm{I}_2 \mid \mathtt{f}^n(\mathrm{I}_1, \ldots, \mathrm{I}_n) \mid \mathtt{split}(\mathrm{I}) & \text{index terms} \\
 & & \mid \mathtt{inl}(\mathrm{I}) \mid \mathtt{inr}(\mathrm{I}) \mid \langle \mathrm{I}_1, \mathrm{I}_2 \rangle \mid \pi_1(\mathrm{I}) \mid \pi_2(\mathrm{I}) & \\
\phi & ::= & \emptyset \mid \phi, a : \gamma \mid \phi, \mathrm{I} & \text{index contexts} \\
\rho & ::= & \epsilon \mid \rho \circ \langle a \mapsto \mathrm{I} \rangle & \text{index substitutions}
\end{array}
$$

- We need to provide index terms that can be used both in types and terms.
- Index terms are sorted and sorts can depend on index terms.
- Constant and function symbols $\mathcal{C}$ are equipped with a signature.
- We also need to define index contexts and substitutions.

Sorting rules for the Index language

$$\frac{\phi(a) = \gamma}{\phi \vdash a : \gamma}\ (v.sort) \qquad \frac{}{\phi \vdash 0 : o}\ (0.sort) \qquad \frac{\phi \vdash \mathrm{I} : o}{\phi \vdash \mathrm{I} + 1 : o}\ (s.sort)$$

$$\frac{\mathcal{S}(\mathtt{C}^n) = \Pi\vec{a}.\gamma_1 \times \cdots \times \gamma_n \to \gamma \quad \phi \vdash \mathrm{I}_k : \gamma_k \quad (1 \le k \le n)}{\phi \vdash \mathtt{C}^n(\mathrm{I}_1, \ldots, \mathrm{I}_n) : \gamma}\ (c.sort)$$

$$\frac{\phi \vdash a : \{a_1 : \gamma \mid \mathrm{I}\}}{\phi \vdash a : \gamma}\ (v.sub)$$

$$\frac{\phi \vdash \mathrm{I} : \gamma \quad \phi, a : \gamma \vdash \mathrm{I}_1 : o \quad \phi \models \mathrm{I}_1\langle a \mapsto \mathrm{I}\rangle}{\phi \vdash \mathrm{I} : \{a : \gamma \mid \mathrm{I}_1\}}\ (c.sub)$$

The intended model of our index term language is an Herbrand-Gödel equational system $\mathcal{E}$ over natural numbers. I.e. $\phi \models \mathrm{I}$ whenever $\mathcal{E}(\phi) \models_{HG} \mathcal{E}(\mathrm{I})$

Type and Term Language Syntax

- Types are given by the following grammar:

$$\sigma, \tau \quad ::= \quad \mathtt{Nat}[I] \mid \mathbf{1} \mid !_{a:\gamma}\sigma \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid \Pi a : \gamma.\sigma \mid \Sigma a : \gamma.\sigma \qquad \text{Types}$$

- We use a Wadler style linear lambda calculus with patterns. The syntax is given by the following grammar:

$$\begin{array}{rcll} p & ::= & * \mid x \mid x \otimes x \mid !(x) & \text{patterns} \\ t & ::= & * \mid x_\rho \mid \mathtt{0}[I] \mid \mathtt{s}[I](t) \mid \lambda p.t \mid tu \mid !_{a:\gamma}(t) \mid t \otimes t & \text{terms} \\ & & \mid \ \mathtt{case}\ t\ \mathtt{of}\ \mathtt{0}[a] \mapsto u \mid \mathtt{s}[a]\ (n) \mapsto v \mid \mathtt{fix}\ x.t & \end{array}$$

- Every constructor comes with a particular type signature. In particular:

$$\mathcal{S}(\mathtt{0}) = \Pi a : o.\mathtt{Nat}[0] \qquad \mathcal{S}(\mathtt{s}) = \Pi a : o.\mathtt{Nat}[a] \rightarrow \mathtt{Nat}[a+1]$$

Typing rules - 1

Multiplicative rules:

$$\frac{}{\phi; x : \sigma \vdash x_{\epsilon} : \sigma}\ (Ax) \qquad \frac{\phi; \Gamma, p : \sigma \vdash t : \tau}{\phi; \Gamma, p : \sigma \vdash \lambda p.t : \sigma \multimap \tau}\ (\multimap)$$

$$\frac{\phi; \Gamma \vdash t : \sigma \quad \phi \models \sigma \equiv \tau}{\phi; \Gamma \vdash t : \tau}\ (\equiv) \qquad \frac{\phi; \Gamma \vdash t : \sigma \multimap \tau \quad \phi; \Delta \vdash u : \sigma}{\phi; \Gamma, \Delta \vdash tu : \tau}\ (Ap)$$

$$\frac{\phi; \Gamma, x : \sigma, y : \tau \vdash t : \mu}{\phi; \Gamma, x \otimes y : \sigma \otimes \tau \vdash t : \mu}\ (\otimes L) \qquad \frac{\phi; \Gamma \vdash t : \sigma \quad \phi; \Delta \vdash u : \tau}{\phi; \Gamma, \Delta \vdash t \otimes u : \sigma \otimes \tau}\ (\otimes R)$$

Pattern matching rules:

$$\frac{}{\phi; \vdash \mathtt{0}[0] : \mathtt{Nat}[0]}\ (0) \qquad \frac{\phi \vdash \mathrm{I} : o \quad \phi; \Gamma \vdash t : \mathtt{Nat}[\mathrm{I}]}{\phi; \Gamma \vdash \mathtt{s}[\mathrm{I}](t) : \mathtt{Nat}[\mathrm{I} + 1]}\ (s)$$

$$\frac{\phi; \Gamma \vdash t : \mathtt{Nat}[\mathrm{I}] \quad \phi, a : o, 0 \doteq \mathrm{I}; \Delta \vdash u : \sigma \quad \phi, a : o, a + 1 \doteq \mathrm{I}; \Delta, n : \mathtt{Nat}[a] \vdash v : \sigma}{\phi; \Gamma, \Delta \vdash \ \mathtt{case}\ t\ \mathtt{of}\ \mathtt{0}[a] \mapsto u \mid \mathtt{s}[a]\ (n) \mapsto v : \sigma}$$

Typing rules - 2

Exponential Rules:

$$\frac{\phi;\Gamma\vdash t:\tau}{\phi;\Gamma,x:!_{a:\gamma}\sigma\vdash t:\tau}\;(w)\qquad\frac{\phi;\Gamma,y:\sigma[a\mapsto\mathrm{I}]\vdash t:\tau\quad\phi\vdash\mathrm{I}:\gamma}{\phi;\Gamma,!x:!_{a:\gamma}\sigma\vdash t[x_{\langle a\mapsto\mathrm{I}\rangle}/y]:\tau}\;(d)$$

$$\frac{\phi;\Gamma,y:!_{a:\gamma_1}\sigma,z:!_{a:\gamma_2}\sigma\vdash t:\tau}{\phi;\Gamma,x:!_{a:\gamma_1+\gamma_2}\sigma[a\mapsto\mathtt{split}(a)]\vdash t[x_{\langle a\mapsto\mathtt{split}(a)\rangle}/y,z]:\tau}\;(c)$$

$$\frac{\phi;\Gamma,!y:!_{a_1:\gamma_1}!_{a_2:\gamma_2}\sigma\vdash t:\tau}{\phi;\Gamma,!x:!_{a:\gamma_1\times\gamma_2}\sigma[a_1\mapsto\pi_1(a),a_2\mapsto\pi_2(a)]\vdash t[x_{\langle a_1\mapsto\pi_1(a),a_2\mapsto\pi_2(a)\rangle}/y]:\tau}\;(g)$$

$$\frac{\phi,a:\gamma;x_1:\sigma_1,\ldots,x_n:\sigma_n\vdash t:\tau}{\phi;!x_1:!_{a:\gamma}\sigma_1,\ldots,!x_n:!_{a:\gamma}\sigma_n\vdash!_{a:\gamma}t:!_{a:\gamma}\tau}\;(p)$$

Some examples.

Clearly we have natural numbers that are decorated in a trivial way:

$$;\vdash \mathtt{0}[0] : \mathtt{Nat}[0] \qquad \cdots \qquad ;\vdash \mathtt{s}[0+1](\mathtt{s}[0]\mathtt{0}[0]) : \mathtt{Nat}[(0+1)+1]$$

Consider the following sorts:

$$\gamma_1 = \{a : o \mid \mathtt{even}(a)\} \qquad \gamma_2 = \{a : o \mid a \doteq 0 \vee \mathtt{odd}(a)\}$$

Then we have a term as:

$$\lambda !x.x_{\langle a \mapsto \mathtt{split}(a)\rangle} \otimes x_{\langle a \mapsto \mathtt{split}(a)\rangle}$$

with type

$$!_{a:\gamma_1+\gamma_2}(\mathtt{Nat}[a] \multimap \mathtt{Nat}[a+1]) \multimap (\mathtt{Nat}[a] \multimap \mathtt{Nat}[a+1]) \otimes (\mathtt{Nat}[a] \multimap \mathtt{Nat}[a+1])$$

Such a term can be applied only to terms acting as successor for zero but we keep more informations.

What we expect…future works

- We are working to prove that the present framework well behaves, i.e. it enjoys some standard properties, e.g. substitution properties. Note that they can turn to be non trivial in such a context.
- We need to develop a general form of soundness property relative to the involved constraint language. In particular, we need to extend to this framework Hofmann’s realizability technique.
- Conversely, we expect to obtain a proof of the intensional completeness in terms of expressivity with respect of the considered language.
- If we succeed in the above points, we would then consider the remaining constructions, in particular fixpoints. We expect that while in DML they are treated in a straightforward way, here they involve more difficulties.
- Finally, we plan to explore complex large examples in this framework.

Thanks!