Implicit Computational Complexity: An Introduction to Non-size-increasing Computation

Ugo Dal Lago

Dipartimento di Scienze dell'Informazione
Università di Bologna

BISS, March 10th 2006

Outline

Motivations

LFPL_ω

LFPL_T

Conclusions

Extensional vs. Intensional

- Many systems in ICC are both intensionally sound and extensionally complete w.r.t. a given complexity class C:
  - Any **program** can be executed according to the definition of C;
  - Any **function** in C is representable.
- But what does **representable** mean?
  - A function $f : \{0,1\}^* \to \{0,1\}^*$ is representable if **there is** a program p which computes f.
  - But there are many programs computing the same function...
- This is definitely a mismatch.

An Example - Sorting

- Let $sort : \mathbb{N}^* \rightarrow \mathbb{N}^*$ be a function which receives as input a finite sequence l of natural numbers and outputs an non-decreasing permutation of l.
- We can distinguish at least three different polynomial time algorithms computing $sort$:
  - First of all, we can iterate a conditional swapping operation a quadratic number of times, in the style of **BubbleSort**.
  - We can iterate an insertion algorithm a linear number of times. The insertion algorithm is itself defined iteratively and takes a linear amount of time. This algorithm is known as **InsertionSort**.
  - We can partition the input sequence l into two subsequences f and s such that any element of f is smaller or equal to any element of s. We then apply recursively the same algorithm to f and s and concatenate the two results. This algorithm is known as **QuickSort**.

An Example - Sorting

- **BubbleSort** and **InsertionSort** can be written in a functional programming language, provided it allows some form of iteration.
- While most of the systems capturing polynomial time admit **BubbleSort** as a legal definition, many of them do not allow nested iterations. As a consequence, **InsertionSort** is usually rejected.
- The situation is even worse for **QuickSort**, because recursion is not structural and the algorithm being polytime critically depends on size considerations about the partition step.

Why is Nested Recursion Prohibited?

- Because it **can possibly** lead to an exponential behavior.
- Consider the following program:

$$
\begin{aligned}
\mathrm{double}(\varepsilon) &= \varepsilon \\
\mathrm{double}(0 \cdot t) &= 0 \cdot 0 \cdot \mathrm{double}(t) \\
\mathrm{double}(1 \cdot t) &= 1 \cdot 1 \cdot \mathrm{double}(t) \\
\mathrm{exp}(\varepsilon) &= 0 \\
\mathrm{exp}(0 \cdot t) &= \mathrm{double}(\mathrm{exp}(t)) \\
\mathrm{exp}(1 \cdot t) &= \mathrm{double}(\mathrm{exp}(t))
\end{aligned}
$$

- Clearly $\mathrm{exp}(t) = 0^{2^{|t|}}$.
- Many ICC systems (safe recursion, ramified recursion, light affine logic, etc.) do not allow nested recursion.

Nested Recursion can Be Benign

- Consider the following slight variation on the previous program:

$$
\begin{aligned}
\mathrm{switch}(\varepsilon) &= \varepsilon \\
\mathrm{switch}(0 \cdot t) &= 1 \cdot \mathrm{switch}(t) \\
\mathrm{switch}(1 \cdot t) &= 0 \cdot \mathrm{switch}(t) \\
\mathrm{parity}(\varepsilon) &= 0 \\
\mathrm{parity}(0 \cdot t) &= \mathrm{switch}(\mathrm{parity}(t)) \\
\mathrm{parity}(1 \cdot t) &= \mathrm{switch}(\mathrm{parity}(t))
\end{aligned}
$$

- Observe $\mathrm{parity}(t) = 0$ if $|t|$ is even and $\mathrm{parity}(t) = 1$ if $|t|$ is odd.
- There is not any exponential blowup anymore.
- Why? switch, as opposed to double, is **non-size increasing**!

LFPL$_\omega$ programs

- **Types**: booleans (B), lists (L(A)), binary trees (T(A)), products ($A \otimes B$), disjoint union ($A + B$), resource type ($\diamond$). In examples: $\mathrm{N} = \mathrm{B} \otimes \ldots \otimes \mathrm{B}$. (32 times)
- **Signatures**: mapping of function symbols f to "arities": $\Sigma(f) = A_1, A_2, \ldots, A_n \to B$, e.g., $\mathrm{append} : \mathrm{L}(\mathrm{N}), \mathrm{L}(\mathrm{N}) \to \mathrm{L}(\mathrm{N})$.
- **Programs**: Signature + for each function symbol f with $\Sigma(f) = A_1, A_2, \ldots, A_n \to B$ a term e_f of type B containing free variables $x_1 : A_1, \ldots, x_n : A_n$. The term e_f may contain calls to f and other functions declared in Σ.

Terms

They are built up from function calls, constructors, and pattern matching like in (first order) functional programming with the following exceptions:

- Constructors of recursive types take an extra argument of type $\diamond$ (unless they are nil):

$$\mathsf{cons}(e_1^{\diamond}, e_2^{A}, e_3^{\mathsf{L}}(A)) : \mathsf{L}(A)$$
$$\text{match } e_1^{\mathsf{L}}(A) \text{ with nil} \Rightarrow e_2^{C} \mid \mathsf{cons}(x^{\diamond}, y^{A}, z^{\mathsf{L}(A)})) \Rightarrow e_3^{C}$$

  (as always pattern matching binds variables).
- Free and bound variables occur at most once (in the usual sense of affine linear types, e.g. occurrences in different branches of case distinction count only once).
- Variables of type B, $B \otimes B$, N may be used more than once.

Examples

$$\begin{aligned}
\text{append} &: \mathsf{L}(C), \mathsf{L}(C) \to \mathsf{L}(C) \\
\text{append}(\text{nil}, l) &= l \\
\text{append}(\text{cons}(d, h, t), l) &= \text{cons}(d, h, \text{append}(t, l))
\end{aligned}$$

Formally:

$$\begin{aligned}
\text{append}(l_1, l_2) &= \text{match } l_1 \text{ with } = \\
& \quad \text{nil} \Rightarrow l_2 \\
& \mid \quad \text{cons}(d, h, t) \Rightarrow \text{cons}(d, h, \text{append}(t, l_2))
\end{aligned}$$

$$\begin{aligned}
\text{reverse} &: \mathsf{L}(C) \to \mathsf{L}(C) \\
\text{reverse}(\text{nil}) &= \text{nil} \\
\text{reverse}(\text{cons}(d, h, t)) &= \text{append}(\text{reverse}(t), \text{cons}(d, h, \text{nil}))
\end{aligned}$$

Examples - II

$$\begin{aligned}
\text{insert} &: \diamond, C, \mathsf{L}(C) \to L(C) \\
\text{insert}(d, x, \text{nil}) &= \text{cons}(d, x, \text{nil}) \\
\text{insert}(d_1, x, cons(d_2, y, l)) &= \text{let compare}(x, y) \text{ be } (x, y, b) \text{ in} \\
\text{if} \quad b & \quad \text{then cons}(d_1, x, \text{cons}(d_2, y, l)) \\
& \quad \text{else cons}(d_1, y, \text{insert}(d_2, x, l))
\end{aligned}$$

$$\begin{aligned}
\text{sort} &: \mathsf{L}(C) \to \mathsf{L}(C) \\
\text{sort}(\text{nil}) &= \text{nil} \\
\text{sort}(\text{cons}(d, x, l)) &= \text{insert}(d, x, \text{sort}(l))
\end{aligned}$$

Examples - III

$$\begin{aligned}
\mathsf{bst} &: \mathsf{L}(C) \to \mathsf{T}(C) \\
\mathsf{bst}(\mathsf{nil}) &= \mathsf{leaf} \\
\mathsf{bst}(\mathsf{cons}(d, h, t)) &= \mathsf{ins}(d, h, \mathsf{bst}(t))
\end{aligned}$$

$$\begin{aligned}
\mathsf{ins} &: \diamond, C, \mathsf{T}(C) \to \mathsf{T}(C) \\
\mathsf{ins}(d, c, \mathsf{leaf}) &= \mathsf{node}(d, c, \mathsf{leaf}, \mathsf{leaf}) \\
\mathsf{ins}(d_1, c_1, \mathsf{node}(d_2, c_2, l, r)) &= \text{if } c_1 \leq c_2 \text{ then} \\
&\quad \mathsf{node}(d_1, c_2, \mathsf{ins}(d_2, c_1, l), r) \\
&\quad \text{else } \mathsf{node}(d_1, c_2, l, \mathsf{ins}(d_2, c_1, r))
\end{aligned}$$

Examples - IV

$$\begin{aligned}
\mathsf{duplist} &: \mathsf{L}(\diamond \otimes \mathsf{B}) \to \mathsf{L}(\mathsf{B}) \otimes \mathsf{L}(\mathsf{B}) \\
\mathsf{duplist}(\mathsf{nil}) &= \mathsf{nil} \otimes \mathsf{nil} \\
\mathsf{duplist}(\mathsf{cons}(d_1, d_2 \otimes h, t)) &= \mathsf{match}\ \mathsf{duplist}(t)\ \mathsf{with} \\
&\quad u \otimes v \Rightarrow \mathsf{cons}(d_1, h, u) \otimes \mathsf{cons}(d_2, h, v)
\end{aligned}$$

$$\begin{aligned}
\mathsf{twice} &: \mathsf{L}(\diamond \otimes \mathsf{B}) \to \mathsf{L}(\mathsf{B}) \\
\mathsf{twice}(l) &= \mathsf{match}\ \mathsf{duplist}(l)\ \mathsf{with} \\
&\quad u \otimes v \Rightarrow \mathsf{append}(u, v)
\end{aligned}$$

Remark: Function twice duplicates length. There is no definable function that squares or exponentiates length. So, really, $\diamond$ enforces linear growth, not zero growth.

Interpretation of LFPL_ω

- Functions are non-size increasing in standard model:

$$
\begin{aligned}
[\![B]\!] &= \{\mathsf{t}, \mathsf{f}\} \\
[\![\mathsf{L}(A)]\!] &= [\![A]\!]^* \\
[\![A \otimes B]\!] &= [\![A]\!] \otimes [\![B]\!] \\
[\![\diamond]\!] &= \{\star\} \\
&\ldots
\end{aligned}
$$

- If $f : A_1, \ldots, A_n \to B$ then $[\![f]\!] : [\![A_1]\!] \times \ldots \times [\![A_n]\!] \to [\![B]\!]$ is defined by least fixpoint.

Expressive power of LFPL_ω

- If $v_1 \in [\![A_1]\!], \ldots, v_n \in [\![A_n]\!]$, then

$$|[\![f]\!](v_1, \ldots, v_n)|_B \leq |v_n|_{A_1} + \ldots + |v_n|_{A_n}$$

  where $|\cdot|_C : [\![C]\!] \rightarrow \mathbb{N} \cup \{\infty\}$.
- So, at least semantically, all definable functions are non-size-increasing.
- The previous observation leads to:

Theorem (Hofmann)

f is representable iff f is computable in time $O(2^{cn})$ for some c (here $n = |w|$). Equivalently f is computable on an $O(n)$ space-bounded Turing machine with unbounded stack [Cook 1972].

LFPL_T

- Structural, Higher-Order Recursion.
- **Types**:

$$A, B ::= \diamond \mid \mathsf{B} \mid A \multimap B \mid A \otimes B \mid \mathsf{L}(A)$$

- **Terms**:

$$\begin{array}{rcl} t, u & ::= & x \mid c \mid \lambda x.t \mid (t)\ u \mid t \otimes u \mid \mathsf{let}\ \ t\ \ \mathsf{be}\ \ x \otimes y\ \ \mathsf{in}\ \ u \mid \\ & & \mathsf{if} \mid \mathsf{iter}_B^{\mathsf{L}(A)}\ \ t\ \ u \end{array}$$

- **Rewriting Rules**:

$$\begin{array}{rcl} (\lambda x.t)u & \to & t[u/x] \\ (\mathsf{iter}_B^{\mathsf{L}(A)}\ \ t\ \ u)\ \mathsf{nil} & \to & u \\ (\mathsf{iter}_B^{\mathsf{L}(A)}\ \ t\ \ u)\ (\mathsf{cons}\ d\ a\ l) & \to & (t)\ d\ a\ (\mathsf{iter}_B^{\mathsf{L}(A)}\ \ t\ \ u)l \\ \mathsf{let}\ \ t_1 \otimes t_2\ \ \mathsf{be}\ \ x \otimes y\ \ \mathsf{in}\ \ u & \to & u[t_1/x, t_2/y] \\ \mathsf{if}\ \mathsf{true}\ u\ v & \to & u \\ \mathsf{if}\ \mathsf{false}\ u\ v & \to & v \end{array}$$

Typing Rules for LFPL_T

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Gamma \vdash t : C}{\Delta, \Gamma \vdash t : C}$$

$$\frac{\Gamma \vdash t : A \multimap B \quad \Delta \vdash u : A}{\Gamma, \Delta \vdash (t)u : B} \qquad \frac{x : A, \Gamma \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}$$

$$\frac{\Gamma \vdash t : A \otimes B \quad x : A, y : B, \Delta \vdash u : C}{\Gamma, \Delta \vdash \text{let} \;\; t \;\; \text{be} \;\; x \otimes y \;\; \text{in} \;\; u : C} \qquad \frac{\Gamma \vdash t : A \quad \Delta \vdash u : B}{\Gamma, \Delta \vdash t \otimes u : A \otimes B}$$

$$\frac{\vdash t : \diamond \multimap A \multimap B \multimap B \quad \vdash u : B}{\vdash \text{iter}_B^{\text{L}(A)} \; t \; u : \text{L}(A) \multimap B} \qquad \frac{}{\vdash \text{true} : \text{B}} \qquad \frac{}{\text{false} : \text{B}}$$

$$\frac{}{\vdash \text{cons} : \diamond \multimap A \multimap \text{L}(A) \multimap \text{L}(A)} \qquad \frac{}{\vdash \text{nil} : \text{L}(A)}$$

$$\frac{}{\vdash \text{if} : \text{B} \multimap A \multimap A \multimap A}$$

Expressive power of LFPL_T

- Some of the previously described examples cannot be catched.
- The calculus is strongly normalizing (it can be embedded into Gödel System T).
- The class of representable functions shrinks:

Theorem (Hofmann)

$f : \{0,1\}^* \to \{0,1\}$ *is representable iff* f *is computable in time* $O(p(n))$ *for some polynomial* p.

- The calculus can be extended with a weak modality ! in the spirit of linear logic and with second-order quantification without losing its nice quantitative properties.

InsertionSort in LFPL_T

$$\vdash \mathit{insert} \;:\; \mathrm{L}(A) \multimap \diamond \multimap A \multimap \mathrm{L}(A)$$
$$\vdash \mathit{sort} \;:\; \mathrm{L}(A) \multimap \mathrm{L}(A)$$

where:

$$\begin{aligned}
\mathit{insert} &= \mathrm{iter}_B^{\mathrm{L}A}\; t^{\diamond\multimap A\multimap B\multimap B}\;\; u^B \\
u &= \lambda d^{\diamond}.\lambda a^{A}.\mathrm{cons}\; d\; a\; \mathrm{nil} \\
t &= \lambda d^{\diamond}.\lambda a^{A}.\lambda f^{B}.\lambda d'^{\diamond}.\lambda a'^{A}. \\
&\qquad \mathrm{let}\;\; (\mathrm{compare}\; a\; a')\;\; \mathrm{be}\;\; a_1 \otimes a_2\;\; \mathrm{in}\;\; \mathit{cons}\; d\; a_1\; (f)\; d'\; a_2 \\
B &= \diamond \multimap A \multimap \mathrm{L}(A)
\end{aligned}$$

InsertionSort in LFPL_T

$$
\begin{aligned}
\mathit{insert}' &= \lambda d^{\diamond}.\lambda a^{A}.\lambda l^{\mathsf{L}(A)}.(\mathit{insert}\ l\ d\ a\ ) \\
\vdash \mathit{insert}' &: \diamond \multimap A \multimap \mathsf{L}(A) \multimap \mathsf{L}(A) \\
\mathit{sort} &= \mathsf{iter}^{\mathsf{L}(A)}_{\mathsf{L}(A)}\ \mathit{insert}'\ \mathsf{nil}
\end{aligned}
$$

Summing Up

- We have presented two programming languages, LFPL_ω and $\mathsf{LFPL}_\mathcal{T}$.
- Every program is non-size-increasing (this is enforced by way of both linearity and $\diamond$).
- Interesting **algorithms** are captured by the systems (for example, **InsertionSort**).

References I

Martin Hofmann.
Linear types and non-size-increasing polynomial time computation.
In *LICS*, pages 464–473, 1999.

Martin Hofmann.
The strength of non-size increasing computation.
POPL, pages 260–269, 2002.

Martin Hofmann.
Linear types and non-size-increasing polynomial time computation.
Information and Computation, 183(1):57–85, 2003.

References II

Martin Hofmann and Steffen Jost.
Static prediction of heap space usage for first-order functional programs.
POPL, pages 185–197, 2003.

Martin Hofmann and Ugo Dal Lago.
Quantitative Models and Implicit Complexity.
FSTTCS, pages 189–2000, 2005.

Questions?