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

$\text{LFPL}_\omega$

$\text{LFPL}_T$

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 $f : \{0,1\}^* \rightarrow \{0,1\}^*$ 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 \( \text{sort} : \mathbb{N}^* \rightarrow \mathbb{N}^* \) 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 \( \text{sort} \):
  
  - First of all, we can iterate a conditional swapping operation a quadratic number of times, in the style of \textbf{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 \textbf{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 \textbf{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:

  \[
  \begin{align*}
  \text{double}(\varepsilon) & = \varepsilon \\
  \text{double}(0 \cdot t) & = 0 \cdot 0 \cdot \text{double}(t) \\
  \text{double}(1 \cdot t) & = 1 \cdot 1 \cdot \text{double}(t) \\
  \exp(\varepsilon) & = 0 \\
  \exp(0 \cdot t) & = \text{double}(\exp(t)) \\
  \exp(1 \cdot t) & = \text{double}(\exp(t)) \\
  \end{align*}
  \]

- Clearly \( \exp(t) = 0^{2^{|t|}} \).
- 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:

\[
\text{switch}(\varepsilon) = \varepsilon \\
\text{switch}(0 \cdot t) = 1 \cdot \text{switch}(t) \\
\text{switch}(1 \cdot t) = 0 \cdot \text{switch}(t) \\
\text{parity}(\varepsilon) = 0 \\
\text{parity}(0 \cdot t) = \text{switch}(\text{parity}(t)) \\
\text{parity}(1 \cdot t) = \text{switch}(\text{parity}(t))
\]

- Observe \(\text{parity}(t) = 0\) if \(|t|\) is even and \(\text{parity}(t) = 1\) 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 ($\mathbb{B}$), lists ($L(A)$), binary trees ($T(A)$), products ($A \otimes B$), disjoint union ($A + B$), resource type ($\diamond$). In examples: $N = \mathbb{B} \otimes \ldots \otimes \mathbb{B}$. (32 times)

- **Signatures**: mapping of function symbols $f$ to "arities": $\Sigma(f) = A_1, A_2, \ldots, A_n \to B$, e.g.,

append : $L(N), L(N) \to L(N)$.

- **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 $\mathbb{B}$ 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 $\Sigma$. 
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):
  
  \[
  \text{cons}(e_1^\diamond, e_2^A, e_3^L(A)) : L(A)
  \]
  
  match \(e_1^L(A)\) with nil \(\Rightarrow e_2^C \mid \text{cons}(x^\diamond, y^A, z^L(A))\) \(\Rightarrow e_3^C\)

  (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

\[
\begin{align*}
\text{append} & : \ L(C), L(C) \to L(C) \\
\text{append}(\text{nil}, l) & = l \\
\text{append}(\text{cons}(d, h, t), l) & = \text{cons}(d, h, \text{append}(t, l)) \\
\end{align*}
\]

Formally:

\[
\begin{align*}
\text{append}(l_1, l_2) & = \text{match } l_1 \text{ with } = \\
& \quad \text{nil } \Rightarrow l_2 \\
& \quad | \quad \text{cons}(d, h, t) \Rightarrow \text{cons}(d, h, \text{append}(t, l_2)) \\
\end{align*}
\]

\[
\begin{align*}
\text{reverse} & : \ L(C) \to L(C) \\
\text{reverse}(\text{nil}) & = \text{nil} \\
\text{reverse}(\text{cons}(d, h, t)) & = \text{append}(\text{reverse}(t), \text{cons}(d, h, \text{nil}))
\end{align*}
\]
Examples - II

insert : ◊, C, L(C) → L(C)

insert(\(d, x, \text{nil}\)) = \text{cons}(d, x, \text{nil})

insert(\(d_1, x, \text{cons}(d_2, y, l)\)) = \text{let compare}(x, y) \text{ be } (x, y, b) \text{ in } \begin{cases} \text{if } b \text{ then } \text{cons}(d_1, x, \text{cons}(d_2, y, l)) \\ \text{else } \text{cons}(d_1, y, \text{insert}(d_2, x, l)) \end{cases}

sort : L(C) → L(C)

sort(\text{nil}) = \text{nil}

sort(\text{cons}(d, x, l)) = \text{insert}(d, x, \text{sort}(l))
Examples - III

\[
\begin{align*}
\text{bst} & : \ L(C) \rightarrow T(C) \\
\text{bst}(\text{nil}) & = \text{leaf} \\
\text{bst}(\text{cons}(d, h, t)) & = \text{ins}(d, h, \text{bst}(t)) \\

\text{ins} & : \ \diamond, C, T(C) \rightarrow T(C) \\
\text{ins}(d, c, \text{leaf}) & = \text{node}(d, c, \text{leaf}, \text{leaf}) \\
\text{ins}(d_1, c_1, \text{node}(d_2, c_2, l, r)) & = \text{if } c_1 \leq c_2 \text{ then} \\
& \quad \text{node}(d_1, c_2, \text{ins}(d_2, c_1, l), r) \\
& \text{else } \text{node}(d_1, c_2, l, \text{ins}(d_2, c_1, r))
\end{align*}
\]
Examples - IV

duplist : L(⋄ ⊙ B) → L(B) ⊙ L(B)

duplist(nil) = nil ⊙ nil

duplist(cons(d_1, d_2 ⊙ h, t)) = match duplist(t) with
    u ⊙ v ⇒ cons(d_1, h, u) ⊙ cons(d_2, h, v)

twice : L(⋄ ⊙ B) → L(B)

twice(l) = match duplist(l) with
    u ⊙ v ⇒ append(u, v)

Remark: Function twice duplicates length. There is no definable function that squares or exponentiates length. So, really, ⋄ enforces linear growth, not zero growth.
Interpretation of LFPL$_\omega$

- Functions are non-size increasing in standard model:

\[
[B] = \{t, f\}
\]
\[
[L(A)] = [A]^\ast
\]
\[
[A \otimes B] = [A] \otimes [B]
\]
\[
[\diamond] = \{\star\}
\]

\[
\ldots
\]

- 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 \( \nu_1 \in [A_1], \ldots, \nu_n \in [A_n] \), then

\[
\| \langle f \rangle (\nu_1, \ldots, \nu_n) \|_B \leq \| \nu_n |_{A_1} + \cdots + \| \nu_n |_{A_n}
\]

where \( | \cdot |_C : [C] \to \mathbb{N} \cup \{\infty\} \).

- 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].
\(\text{LFPL}_T\)

- Structural, Higher-Order Recursion.
- Types:
  \[A, B ::= \diamond \mid B \mid A \rightarrow B \mid A \otimes B \mid L(A)\]
- Terms:
  \[t, u ::= x \mid c \mid \lambda x.t \mid (t) u \mid t \otimes u \mid \text{let } t \text{ be } x \otimes y \text{ in } u \mid \text{if } \mid \text{iter}^L(A) t u\]
- Rewriting Rules:
  \[(\lambda x.t)u \rightarrow t[u/x]\]
  \[(\text{iter}^L(A) t u) \text{ nil} \rightarrow u\]
  \[(\text{iter}^L_B t u) (\text{cons } d a l) \rightarrow (t) d a (\text{iter}^L_B t u) l\]
  \[\text{let } t_1 \otimes t_2 \text{ be } x \otimes y \text{ in } u \rightarrow u[t_1/x, t_2/y]\]
  \[
  \text{if true } u v \rightarrow u
  \]
  \[
  \text{if false } u v \rightarrow v
  \]
Typing Rules for LFPLₜ

\[
\begin{align*}
  x : A & \vdash x : A \\
  \Gamma, \Delta & \vdash (t)u : B \\
  \Gamma & \vdash t : A \rightarrow B \\
  \Delta & \vdash u : A \\
  \Gamma, \Delta & \vdash \text{let } t \text{ be } x \otimes y \text{ in } u : C \\
  \Gamma & \vdash t : A \otimes B \\
  x : A, y : B, \Delta & \vdash u : C \\
  \Gamma, \Delta & \vdash t \otimes u : A \otimes B \\
  \Gamma & \vdash \text{true} : B \\
  \Gamma & \vdash \text{false} : B \\
  \Gamma & \vdash \text{cons} : \diamond \rightarrow A \rightarrow L(A) \rightarrow L(A) \\
  \Gamma & \vdash \text{nil} : L(A) \\
  \Gamma & \vdash \text{if} : B \rightarrow A \rightarrow A \rightarrow A
\end{align*}
\]
Expressive power of LFPL$_T$

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

**Theorem (Hofmann)**

$f : \{0, 1\}^* \rightarrow \{0, 1\}$ 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 $\tau$

\[ \vdash \text{insert} : \text{L}(A) \circ \circ \circ A \circ \circ \text{L}(A) \]

\[ \vdash \text{sort} : \text{L}(A) \circ \circ \text{L}(A) \]

where:

\[
\begin{align*}
  \text{insert} & \quad = \quad \text{iter}^\text{LA}_B \ t^{\circ \circ A \circ \circ B \circ \circ B} \ u^B \\
  u & \quad = \quad \lambda d^{\circ}. \lambda a^A. \text{cons} \ d \ a \ \text{nil} \\
  t & \quad = \quad \lambda d^{\circ}. \lambda a^A. \lambda f^B. \lambda d'^{\circ}. \lambda a'^A. \\
  & \quad \quad \text{let (compare } a \ a') \ \text{be } a_1 \otimes a_2 \ \text{in } \text{cons} \ d \ a_1 \ (f) \ d' \ a_2 \\
  B & \quad = \quad \circ \circ A \circ \circ \text{L}(A)
\end{align*}
\]
**InsertionSort** in LFPL$_T$

\[
insert' = \lambda d^\Diamond. \lambda a^A. \lambda l^{L(A)}. (insert \ l \ d \ a )
\]

\( \vdash insert' : \Diamond \rightarrow A \rightarrow L(A) \rightarrow L(A) \)

\[
sort = iter_{L(A)}^{L(A)} \ insert' \ \text{nil}
\]
Summing Up

- We have presented two programming languages, $\text{LFPL}_\omega$ and $\text{LFPL}_T$.
- 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


Questions?