Programming Languages for Logarithmic Space

Martin Hofmann

LMU München

Geocal '06

Luminy, 13. Feb. 2006

Outline

- Preliminaries (LOGSPACE, ICC)
- Mairson–Möller-Neergaard's BC-minus.
- An LFPL-based language for LOGSPACE.
- Transitive Closure Logic.
- A While language for a subset of LOGSPACE.
- Conclusions.

Space-bounded Turing machines

A *Turing machine for sublinear space* (TMS) has one or more desginated input tape(s) which can only be read from not written to.

It also has one or more designated output tape(s) that can only be written to and not read from.

In addition it has a number of worktapes that it can write to and read from.

Σ is a finite alphabet, e.g., $\Sigma = \{0, 1\}$.

A TMS is $s(n)$-space bounded if on any input w (or $\vec{w}$ presented on the input tape it will halt by reaching an accepting or rejecting state and during the entire computation only the first $s(n)$ cells of each worktape will have been used.

An arbitrarily large portion of the output tape may have been written to.

LOGSPACE

A language $A \subseteq \Sigma^*$ is in the complexity class L or LOGSPACE if there exists an $O(\log(n))$-space bounded deterministic TMS which accepts A.

A function $f : \Sigma^* \to \Sigma^*$ is in FL or FLOGSPACE if there exists a $O(\log(n))$- space bounded deterministic TMS which computes f.

NL (or NLOGSPACE) is defined like L but with nondeterministic TMS.

POLYLOGSPACE is defined like LOGSPACE but with $O(\log^k(n))$ space bound.

Classical results

NL is contained in POLYLOGSPACE, in fact $O(\log^2(n))$ (Savitch)

Reachability in directed graphs is complete for NL.

NL is closed under complement (Immerman-Szelepcsenyi).

+,-,*,/ are in FL.

FL is closed under composition.

NL is contained in P (PTIME)

FL is contained in FP

Reachability in undirected graphs is in L (Reingold).

Most important open problem: is L=P ?

Implicit Computational Complexity

Characterisation of complexity classes with programming languages / logics / type systems.

New independent resource-free characterisations of complexity classes

Better understanding of the strength of programming idioms and logical systems.

Alternative route to separations, at least relative ones.

Proofs often lead to more resource-efficient evaluation strategies.

Instances of ICC

For polytime: many instances:

Safe recursion, bounded recursion, LLL, $\mathsf{LFPL}_{\mathrm{poly}}$, BLL, Fixpoint logic, Datalog.

Also for other classes:

SO$\exists$:NP, ELL: Elementary, LFPL_{ω}: EXPTIME, ...

ICC for LOGSPACE

ICC for LOGSPACE is particularly interesting because:

- LOGSPACE is the complexity class of "large data" that does not fit into the store entirely.
- Separating L from P or even NP seems more accessible than separating P from NP or such like.
- Programming LOGSPACE Turing Machines is awkward due to complicated composition. Programming language support useful.

Yet, surprisingly little existing work.

Existing characterisations of LOGSPACE

Unary safe recursion

Read-only cons-free programs

Safe linear recursion (Mairson–Möller-Neergaard)

Transitive Closure Logic (Immerman)

This talk:

An LFPL version of Moeller-Neergaard

A programming language based on transitive closure logic.

The Mairson–Möller-Neergard system

Safe recursion (Bellantoni-Cook):

Variant of primitive recursion for lists.

Distinguish between safe and normal variables.

Results of recursive calls to be plugged into safe argument positions only

Variables that are recursed on must be marked normal.

Example:

Legal:

```
append([];l) = l
append(h::t;l) = h :: append(t;l)
multi([]) = [tt]
multi(h::t, l) = append(l;multi(t))
```

Illegal:

```
f([]) = [tt]
f(h::t) = append(f(t);f(t))
```

Bellantoni-Cook result: all Polytime functions definable.

Invariant: $|f(\vec{x};\vec{y})| \leq p(|\vec{x}|) + \max|\vec{y}|$

Mairson and Möller-Neergard

Like Safe Recursion (BC) but safe variables to be used at most once: *linearly*.

So, append is ok, but

$$\texttt{parity}(h :: t; l) = \texttt{nth}(h :: t, l) \oplus \texttt{parity}(t; l)$$

is not ok.

Invariant: every bit of $f(\vec{x}; \vec{y})$ depends only on one specific bit of $\vec{y}$.

Intuition: safe input is like a function mapping queries to answers.

A (affine) linear functional language

- *Types:* booleans (B), lists (L(A)), binary trees (T(A)), products ($A \otimes B$), disjoint union ($A + B$), resource type ($\diamond$).

  In examples: N = B $\otimes \cdots \otimes$ B (32 times), type variables.

- *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 $\Sigma(f) = A_1, \ldots, A_n \to B$ 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

built up from function calls, constructors, and pattern matching like in functional programming with the following exceptions:

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

$$\mathsf{cons}(e_1^{\Diamond}, e_2^{A}, e_3^{\mathsf{L}(A)}) : \mathsf{L}(A)$$

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

  (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 $\mathsf{B}, \mathsf{B} \otimes \mathsf{B}, \mathsf{N}, \ldots$ may be used more than once.

Examples

$\mathtt{append} : \mathsf{L}(C), \mathsf{L}(C) \to \mathsf{L}(C)$

$\mathtt{append}(l, \mathsf{nil}) = l$

$\mathtt{append}(\mathsf{cons}(d, h, t), l) = \mathsf{cons}(d, h, \mathtt{append}(t, l))$

Formally:

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

$\mathtt{reverse} : \mathsf{L}(C) \to \mathsf{L}(C)$

$\mathtt{reverse}(\mathsf{nil}) = \mathsf{nil}$

$\mathtt{reverse}(\mathsf{cons}(d, h, t)) = \mathtt{append}(\mathtt{reverse}(t), \mathsf{cons}(d, h, \mathsf{nil}))$

$$\texttt{insert} : \diamond, C, \mathsf{L}(C) \to \mathsf{L}(C)$$

$$\texttt{insert}(d, x, \mathsf{nil}) = \mathsf{cons}(d, x, \mathsf{nil})$$

$$\texttt{insert}(d_1, x, \mathsf{cons}(d_2, y, l)) = \mathsf{let}\ (x, y, b) = \texttt{compare}(x, y)\ \mathsf{in}$$

$$\quad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{cons}(d_1, x, \mathsf{cons}(d_2, y, l))$$

$$\quad\ \mathsf{else}\ \mathsf{cons}(d_1, y, \texttt{insert}(d_2, x, l))$$

$$\texttt{sort} : \mathsf{L}(C) \to \mathsf{L}(C)$$

$$\texttt{sort}(\mathsf{nil}) = \mathsf{nil}$$

$$\texttt{sort}(\mathsf{cons}(d, x, l)) = \texttt{insert}(d, x, \texttt{sort}(l))$$

$$\texttt{bst} : \mathsf{L}(C) \to \mathsf{T}(C)$$

$$\texttt{bst}(\mathsf{nil}) = \mathsf{leaf}$$

$$\texttt{bst}(\mathsf{cons}(d, h, t)) = \texttt{ins}(d, h, \texttt{bst}(t))$$

$$\texttt{ins} : (\diamond, C, \mathsf{T}(C)) \to \mathsf{T}(C)$$

$$\texttt{ins}(d, c, \mathsf{leaf}) = \mathsf{node}(d, c, \mathsf{leaf}, \mathsf{leaf})$$

$$\texttt{ins}(d_1, c_1, \mathsf{node}(d_2, c_2, l, r)) = \mathsf{if}\ c_1 \leq c_2\ \mathsf{then}$$

$$\mathsf{node}(d_1, c_2, \texttt{ins}(d_2, c_1, l), r)$$

$$\mathsf{else}\ \mathsf{node}(d_1, c_2, l, \texttt{ins}(d_2, c_1, r))$$

$$\texttt{head\_tail} : \mathsf{L}(C) \to C \otimes \diamond \otimes \mathsf{L}(C)$$

$$\texttt{head\_tail}(\mathsf{nil}) = \texttt{head\_tail}(\mathsf{nil})$$

$$\texttt{head\_tail}(\mathsf{cons}(d, x, l)) = d \otimes x \otimes l$$

$\texttt{duplist} : \mathsf{L}(\Diamond \otimes \mathsf{B}) \to \mathsf{L}(\mathsf{B}) \otimes \mathsf{L}(\mathsf{B})$

$\texttt{duplist}(\mathsf{nil}) = \mathsf{nil} \otimes \mathsf{nil}$

$\texttt{duplist}(\mathsf{cons}(d_1, d_2 \otimes h, t)) = \mathsf{match}\ \texttt{duplist}(t)\ \mathsf{with}$

$u \otimes v \Rightarrow \mathsf{cons}(d_1, h, u) \otimes \mathsf{cons}(d_2, h, v)$

$\texttt{twice} : \mathsf{L}(\Diamond \otimes \mathsf{B}) \to \mathsf{L}(\mathsf{B})$

$\texttt{twice}(l) = \mathsf{match}\ \texttt{duplist}(l)\ \mathsf{with}$

$u \otimes v \Rightarrow \mathsf{append}(u, v)$

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

Results on LFPL

LFPL captures the complexity class $E = TIME(2^{O(n)})$.

LFPL with higher-order functions captures the class $EXP = TIME(2^{n^{O(1)}})$.

LFPL with structural recursion and higher-order functions captures Polytime.

LFPL with structural recursion and first-order functions captures polytime and simultaneously linear space.

LFPL with first-order functions and tail recursion only captures linear space.

LFPL with structural recursion and multiple use of variables in boolean context captures Polyspace.

Important intuition gleaned from Moeller Neergaard

Distinguish *large values*, i.e., those of size $O(\mathit{poly}(n))$ where $n=$ size of the input.

. . . from *small values*, i.e., those of size $O(\log(n))$.

Our (Uli Schoepp, Jan Johannsen, MH) idea:

Use LFPL and represent large values as functions $\mathsf{L}(\mathsf{B}) \multimap \mathsf{B} \otimes \mathsf{L}(\mathsf{B})$; small values as $\mathsf{L}(\mathsf{B})$.

To be precise, large values should be represented as pairs $\mathsf{L}(I) \otimes (\mathsf{L}(\mathsf{B}) \multimap \mathsf{B} \otimes \mathsf{L}(\mathsf{B}))$. First component gives the length in unary.

LOGSPACE in LFPL

Theorem: Suppose that P is a first-order LFPL program containing an uninterpreted function symbol $\mathtt{l} : \mathsf{L}(\mathsf{B}) \multimap \mathsf{B} \otimes \mathsf{L}(\mathsf{B})$ and a defined function symbol $\mathtt{result} : \mathsf{L}(I), \mathsf{L}(\mathsf{B}) \to \mathsf{B}$.

Consider the function $F : \mathsf{B}^* \to \mathsf{B}^*$ defined as follows:

$F(l)_i = \mathtt{result}(\star^{lh(l)}, i)$ whenever $\mathtt{l}(j) = (l_j, j)$.

Then F is in FLOGSPACE and any FLOGSPACE function is of that form.

Intuitive proof

Translate programs into malloc()-free C, in particular $\diamond$ as `void *`.

```
typedef ... C;

list_C cons(void *d, C hd, list_C tl)
{
  (list_C)d->hd = hd;
  (list_C)d->tl = tl;
  return (list_C)d;
}
```

Show using logical relation that for well-typed programs this translation yields the correct result.

Linearity is really needed: append(l, l) creates a circular list.

Proof, cont'd plus morale

Initial number of Diamonds $= \log(n)$, so, all intermediate data structures will have logarithmic size.

Formal proof uses geometry of interaction, see Uli Schoepp's talk.

Note: Composition of such second-order LFPL programs is easy, just substitute functions.

Could be made precise in a higher-order version with explicit function types.

Supports time-space tradeoff in compilation: you can either compile to LOGSPACE (slow but space-efficient) or, using memo tables for functions to PTIME (fast, but uses lots of space).

Relationship to other function algebras

Bellantoni'92 shows that LOGSPACE is captured by allowing "cons" only in the form $1 ::?$, not $0 ::?$. In this way, all intermediate lists are unary (conceptually of type $\mathsf{L}(I)$) and represent logarithmic information.

Neil Jones' "read-only cons-free" programs also capture LOGSPACE: in that system in-place modification of a list is allowed but cons is not.

Jones also mentions possibility of time-space tradeoff.

Summary

We have introduced a variant of LFPL that captures LOGSPACE.

Large values are represented as functions.

LFPL shares some properties with Jones' languages but is fully functional, concrete programs perhaps easier to write.

Extension with higher-order functions for intermediate large values would be desirable. Best understood via GoI.

We have surveyed other approaches to ICC for LOGSPACE among them Mairson, Möller- Neergaard, and Bellantoni.

Transitive closure logic

In Finite Model Theory computations are represented as formulas defining the property to be decided.

The input is represented as a relational structure, e.g., the individuals are the nodes of a graph and the edge relation is represented as a binary relation.

If one wants to name outgoing and incoming edges of a regular graph (regular = all nodes have the same degree) one can use a family of relations $E_{ab}(x, y)$ which signifies that the a-th edge at node x equals the b-th edge at node y, hence in particular x and y are linked.

Formulas are built from propositional connectives, first-order quantifiers, and the following transitive closure construct:

Transitive closure operator

If $\phi(\vec{x}, \vec{y})$ is a formula in $2k$ variables then so is the formula $\mathit{TC}(\vec{x}, \vec{y}.\phi(\vec{x}, \vec{y}))(\vec{u}, \vec{v})$ in which the variables $\vec{x}$ and $\vec{y}$ are bound, whereas $\vec{u}$ and $\vec{v}$ are free.

Its semantics is the transitive closure of the binary relation on D^k defined by ϕ.

Deterministic transitive closure is the restriction of TC to partial functions, i.e., formulas ϕ such that $\phi(\vec{x}, \vec{y}), \phi(\vec{x}, \vec{y}') \Rightarrow \vec{y} = \vec{y}'$.

One can define deterministic transitive closure for all formulas and thus avoid semantic side conditions:

$$\mathit{DTC}(\vec{x}, \vec{y}.\phi(\vec{x}, \vec{y}))(\vec{u}, \vec{v}) = \mathit{TC}(\vec{x}, \vec{y}.\phi(\vec{x}, \vec{y}) \wedge \forall \vec{y}'.\phi(\vec{x}, \vec{y}') \Rightarrow \vec{y} = \vec{y}')(\vec{u}, \vec{v})$$

Transitive Closure Logic: ordering

Usually, transitive closure logic assumes a primitive binary relation $\leq$ which must be interpreted as a linear ordering on individuals.

E.g. lexicographic ordering of bitstrings.

With the order it makes no difference whether edges are numbered or not and we have Immerman's result:

Transitive closure logic captures NLOGSPACE

Deterministic transitive closure logic captures LOGSPACE

Proof of Immerman's theorem

The proof encodes log-size worktapes as individuals. We can define the successor relation on individuals by

$$x = S(y) \Longleftrightarrow x < y \wedge \forall z.x < z \Rightarrow y \leq z$$

Now, we can define addition by transitive closure:

$$x = y + z \Longleftrightarrow \mathit{DTC}((a,b),(c,d).c = S(a) \wedge d = S(b))((y,0),(x,z))$$

With addition we can define $\lambda x.2x, \lambda x.x/2$ etc. and thus extract individual bits.

Now, we can define the one step relation of a LOGSPACE TM and iterate it using *DTC* or *TC*.

Evaluating TC in LOGSPACE

Quantifiers are evaluated by exhaustive search.

Transitive closure by iteration; maintaining a log-size counter to detect failure.

Leave out the ordering?

Without the order DTC becomes too weak:

Let C_k be a circle of length k (undirected)

Let $2C_k$ be $\{0,1\} \times C_k$ with edges given by $(i,u) \leftrightarrow (j,v)$ if $u \leftrightarrow v$ in C_k.

Grädel-McColm '92: DTC without order cannot distinguish two copies of $2C_k$ from a single $2C_{2k}$.

Reason: Suppose that $DTC(\vec{x},\vec{y}.\phi(\vec{x},\vec{y}))(\vec{u},\vec{v})$ holds for some $\vec{u},\vec{v}$ in $2C_{2k}$, say, and let S be the smallest set of nodes containing the $\vec{u}$ and closed under partners (if $(i,k) \in S$ then $(1-i,k) \in S$).

Then $\vec{v} \in S$!

To see this, define

$$\alpha(i,u) = \begin{cases} (i,u), \text{ if} (i,u) \in S \\ (1-i,u) \text{ otherwise} \end{cases}$$

Now, suppose $\phi(\vec{u},\vec{u}_1), \phi(\vec{u}_1,\vec{u}_2), \ldots, \phi(\vec{u}_n,\vec{v})$.

We have $\alpha(\vec{u}) = \vec{u}$ so $\phi(\vec{u},\vec{u}_1)$ and $\phi(\vec{u},\alpha(\vec{u}_1))$, so $\vec{u}_1 = \alpha(\vec{u}_1)$ since ϕ is deterministic. And so forth until $\alpha(\vec{v}) = \vec{v}$, so $\vec{v} \in S$.

DTC without order might be more powerful with labelled edges / rotation maps.

Desiderata for a new language for LOGSPACE

Make precise the intuition: Constant number of graph nodes in memory *only*. Without introducing unwanted weakness such as in DTC.

Idea: Use simple programming language with variables for graph nodes (or other external data).

Use iterators to allow exploration of the entire input.

In such a language the ordinary LOGSPACE graph algorithms (right-hand rule, etc) will be expressible, but those which use log-size extra memory other than for counting might not.

Thus, there is perhaps hope for nontrivial separation results.

Programming language for LOGSPACE: PLL

We fix a set of abstract types corresponding to the input. For simplicity, we assume only one type V, e.g., of graph nodes.

Types then are built up from B (Booleans) and V by cartesian product.

We assume a set of typed relations, e.g., the edge relation $E(x, y)$ of the input graph.

We assume a supply of typed program variables.

Expressions e of boolean type are $\mathsf{tt}, \mathsf{ff}, E(\vec{x}), x{=}y$ where E is an input relation and x, $\vec{x}$, and y are variables of the same type. The only expressions of abstract types are variables.

Statements are given by

$$c ::= c_1; c_2 \mid x := e \mid \texttt{while}(e)c \mid \texttt{if}(b)\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{for}(x)c$$

Semantics of PLL

A program is evaluated relative to a given input structure that assigns a finite set to the abstract base type and interprets the input relations as relations on this set.

An oracle is an infinite sequence of permutations of the base set.

A program state $\sigma = (\eta, o)$ is a valuation η of the variables with type-correct values and an oracle o.

Expressions are interpreted relative to a program state in the usual way, in particular, relation symbols are interpreted according to the input structure.

Statements are interpreted as partial functions on program states.

Assignments and while loops are interpreted as usual...

Semantics of For-Loop

To interpret $\texttt{for}(x)c$ let the current program state be η, o where $o = \pi, o'$ and let $\pi = (v_1, v_2, \ldots, v_n)$.

We then define a sequence of states σ_i for $i = 1, \ldots, n+1$ by $\sigma_1 = (\eta[x{:=}v_1], o')$ and $\sigma_{i+1} = [\![c]\!](\sigma_i[x{:=}v_i])$.

σ_{n+1} then is the resulting program state after executing the for-loop.

In simple terms, the variable x gets assigned all elements of the domain of individuals in an order specified by the first permutation of the current oracle.

It is crucial that this order is neither pre-determined nor necessarily identical through different runs of a for-loop.

It can therefore not be used to define an ordering!

Computation in PLL

A function, predicate, relation, is computed by a PLL program P only if P gives the same (and correct) result *irrespective* of the oracle.

Thus, in order to physically execute PLL programs, any oracle can be used, e.g. one that always uses the same permutation.

Note: Although PLL makes choices during the computation it is neither nondeterministic (there must exist an accepting computation) nor co-nondeterministic (all computations must accept).

The difference is that in PLL a yes/no-answer is required.

PLL's for-loop may be compared to iterators over unordered data in Java or OCaml.

Interpretation of DTC

Proposition: For each DTC-formula $\phi(\vec{x})$ (without ordering) there exists a PLL program with $\vec{x}$ and *result* among its variables so that for any valuation η of the variables $\vec{x}$ upon termination the variables *result* contains tt iff $\phi(\eta)$ holds.

Proof: Use for-loops for quantifiers, while-loop for transitive closure. Given deterministic $\phi(\vec{x}, \vec{y})$ one searches for $\vec{y}$ given $\vec{x}$ using nested for-loops.

Evaluation of PLL in LOGSPACE

Proposition: PLL programs can be evaluated in LOGSPACE.

Proof: Since the meaning is independent of the oracle, any oracle can be chosen, for instance the one induced by the natural ordering. The other parts are direct.

Strength of PLL

Proposition: PLL is strictly stronger than unordered DTC.

Proof1: PLL can decide whether the number of points is even.

Proof2: $2C_{2k}$ and $2 \times 2C_k$ can be distinguished by the following program:

An example program

- Find four points a, b, c, d that form a butterfly pattern.
- Copy a, b, c, d into u, v, w, x.
- Until you reach a, b, c, d again do the following:
  - Find r, s connected to u, v such that $r \neq s$ and distinct from u, v, w, x.
  - Replace u, v, w, x by r, s, u, v.
- Check whether there exists a point not visited by this process. If yes: $2 \times 2C_k$. If no: $2C_{2k}$.

Main Conjecture about PLL

PLL cannot decide reachability in directed or undirected graphs.

In particular, Reingold's recent algorithm for reachability in undirected graphs cannot be implemented in PLL.

Intuition1: Reingold's algorithm requires log-many boolean variables for several things, in particular in order to explore a log-sized neighbourhood of every node.

Intuition2: For any putative PLL program an adversary can modify both input graph (consistently, though) and oracle (without any constraint) to thwart any attempt to decide connectivity.

Summary & Conclusion

- Function algebras, functional programming languages for LOGSPACE: Bellantoni, Jones, Mairson–Möller-Neergaard.
- New language: an LFPL-dialect with large input presented as functions. Connection with GoI $\rightarrow$ Thursday (Uli Schoepp).
- LOGSPACE $\simeq$ Constant number of graph nodes visible.
- Transitive Closure Logic can be seen as one attempt to make that precise, order gets in the way, though.
- New programming language PLL as a better precisation of the "constant number of nodes" intuition.
- PLL strictly stronger than DTC w/o order, conjectured to be weaker than LOGSPACE by adversary argument.

Directions for research

- Prove my conjecture about PLL,
- Linear Logic for LOGSPACE,
- Automating time-space tradeoff for LOGSPACE LFPL (cf. Remark by Neil Jones)
- Add higher-order functions to LOGSPACE languages, see also Uli Schoepp's talk.