

Non size-increasing computation

Martin Hofmann

LMU München

Geocal '06

Luminy, 9.-10. Feb. 2006

Main result of these talks

Description: The system $LFPL_\omega$ is a linearly typed functional language with recursive datatypes and higher-order functions. The growth of definable functions is controlled by an abstract type \diamond elements of which must be present in order to construct a member of a recursive datatype and which become available upon destruction of such a member in a pattern match.

Theorem: A function $f : \mathbb{N} \rightarrow \{0, 1\}$ is in the complexity class $EXPTIME = DTIME(2^{n^{O(1)}})$ if and only if it is representable in $LFPL_\omega$. The restriction of $LFPL_\omega$ to structural recursion captures polynomial time.

Outline

Today:

- Motivation and background
- Definition of $LFPL$ and $LFPL_\omega$
- Proof that structural recursion captures polytime (part I)
- New connective: The “LFPL-bang”.

Tomorrow morning:

- Proof that structural recursion captures polytime (part II)
- Representation of polyspace in $LFPL_\omega$

Tomorrow afternoon:

- Proof that $LFPL_\omega$ programs can be evaluated in $EXPTIME$.

Motivation and Background

1. Programming languages capturing complexity classes.

- Bounded recursion (Cobham, Cook-Urquhart)
- Safe and ramified recursion (Cook-Bellantoni, Leivant, Caseiro)
- Finite model theory (Gurevich, Goerdt)
- Higher-order and type-theoretic extensions (BLL, LLL, SLL, Bellantoni et al, Aehlig-Schwichtenberg, MH)

2. Memory management

- Alias analysis (Morrisett-Walker, Reps-Wilhelm)
- Logic for pointers (O'Hearn-Calcano, Reynolds)
- Pointer analysis (Klarlund et al.)
- In-place update (Shankar, Mackie, Cooper)

Non size-increasing functions

Want to discover statically whether

- the complexity of a function does not “explode” under iteration
- a function can be evaluated in a given amount of space.

Useful (to an extent) answer: isolate statically functions that do not increase the size of their input.

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: $N = B \otimes \dots \otimes B$ (32 times), type variables.

- *Signatures*: mapping of function symbols f to “arities”
 $\Sigma(f) = A_1, A_2, \dots, A_n \rightarrow B$, e.g., $\text{append} : L(N), L(N) \rightarrow L(N)$.
- *Programs*: Signature + for each function symbol f with
 $\Sigma(f) = A_1, \dots, A_n \rightarrow B$ a term e_f of type B containing free variables $x_1:A_1, \dots, x_n:A_n$. 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):

$$\text{cons}(e_1^\diamond, e_2^A, e_3^{L(A)}) : L(A)$$

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

(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 $B, B \otimes B, N, \dots$ may be used more than once.

Examples

$\text{append} : L(C), L(C) \rightarrow L(C)$

$\text{append}(l, \text{nil}) = l$

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

Formally:

$\text{append}(l_1, l_2) = \text{match } l_1 \text{ with nil} \Rightarrow l_2$

| $\text{cons}(d, h, t) \Rightarrow \text{cons}(d, h, \text{append}(t, l_2))$

$\text{reverse} : L(C) \rightarrow L(C)$

$\text{reverse}(\text{nil}) = \text{nil}$

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

$\text{insert} : \diamond, C, L(C) \rightarrow L(C)$

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

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

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

 else $\text{cons}(d_1, y, \text{insert}(d_2, x, l))$

$\text{sort} : L(C) \rightarrow L(C)$

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

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

$$\text{bst} : L(C) \rightarrow T(C)$$

$$\text{bst}(\text{nil}) = \text{leaf}$$

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

$$\text{ins} : (\diamond, C, T(C)) \rightarrow T(C)$$

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

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

$$\quad \text{node}(d_1, c_2, \text{ins}(d_2, c_1, l), r)$$

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

$$\text{head_tail} : L(C) \rightarrow C \otimes \diamond \otimes L(C)$$

$$\text{head_tail}(\text{nil}) = \text{head_tail}(\text{nil})$$

$$\text{head_tail}(\text{cons}(d, x, l)) = d \otimes x \otimes l$$

$$\text{duplist} : L(\diamond \otimes B) \rightarrow L(B) \otimes L(B)$$

$$\text{duplist}(\text{nil}) = \text{nil} \otimes \text{nil}$$

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

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

$$\text{twice} : L(\diamond \otimes B) \rightarrow L(B)$$

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

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

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

Would like to see more examples?

See

http:

[//www.dcs.ed.ac.uk/home/resbnd/prototypes/by_Robert_Atkey/](http://www.dcs.ed.ac.uk/home/resbnd/prototypes/by_Robert_Atkey/)

for more examples and online experiments.

Functions are non-size increasing in standard model

$$\llbracket B \rrbracket = \{\text{tt}, \text{ff}\} \qquad \llbracket L(A) \rrbracket = \llbracket A \rrbracket^*$$

$$\llbracket A \otimes B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket \qquad \llbracket \diamond \rrbracket = \{\star\}$$

...

If $f : A_1, \dots, A_n \rightarrow B$ then $\llbracket f \rrbracket : \llbracket A_1 \rrbracket \times \dots \times \llbracket A_n \rrbracket \rightarrow \llbracket B \rrbracket$ defined as least fixpoint (recursively).

If $v_1 \in \llbracket A_1 \rrbracket, \dots, v_n \in \llbracket A_n \rrbracket$ then

$$|\llbracket f \rrbracket(v_1, \dots, v_n)|_B \leq |v_1|_{A_1} + \dots + |v_n|_{A_n}$$

where $|_ |_C : \llbracket C \rrbracket \rightarrow \mathbb{N} \cup \{\infty\}$, e.g., $|\llbracket u_1, \dots, u_\ell \rrbracket_{L(C)}| = \ell + \sum_i |u_i|_C$ and $|\star|_\diamond = 1$.

NB: Evaluation in a finite model is *sound*. Cf. Gurevich, Goerdts.

Characterisation of definable functions

A function $f : \{0, 1\}^* \rightarrow \{0, 1\}$ is *representable* if there exists a program containing a function symbol $\mathfrak{f} : L(B) \rightarrow B$ with $f = \llbracket \mathfrak{f} \rrbracket$.

Theorem: 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).



$W = L(B^{\otimes c}) \otimes B^{\otimes q}$ represents c work tapes and finite control with q states. $S = B^{\otimes s}$ represents stack symbols. Define helper functions

$$\text{init} : L(B) \rightarrow W$$

$$\text{output} : W \rightarrow L(B)$$

$$\text{run} : W, S \rightarrow W$$

where $\llbracket \text{run} \rrbracket(w, s)$ is the tape inscription+state of the machine when started on one-element stack s .

$$\llbracket \text{run} \rrbracket(w, s) = \begin{cases} w', & \text{if } s \text{ is popped} \\ \llbracket \text{run} \rrbracket(\llbracket \text{run} \rrbracket(w', s_1), s), & \text{if } s_1 \text{ is pushed} \end{cases}$$

(w' = successor configuration)

Only if

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.

Further elaboration

- Read-only types: quest for laxer conditions than linearity which still guarantee that translation to C is correct. ESOP 02 with David Aspinall.
- Inference of \diamond -resource: given linear functional program (without \diamond) return integer linear program whose solutions yield annotations of the program with \diamond . Diploma thesis by Steffen Jost based on earlier work by Dilsun Kırılı.
- Automatic amortised space analysis for functional (POPL03) and object-oriented (ESOP06) programs (Jost & MH).

Higher-order functions

Add type former $A \multimap B$ and terms

$e_1^{A \multimap B} e_2^A : B$ Application; $\text{FV}(e_1) \cup \text{FV}(e_2) = \emptyset$

$\lambda x^A. e^B : A \multimap B$ Abstraction.

$\llbracket A \multimap B \rrbracket = \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket$

Definition: $LFPL_\omega$ is $LFPL$ together with linear function spaces.

Example: Continuation passing style:

$\text{rev} : L(A), L(A) \multimap L(A) \rightarrow L(A)$

$\text{rev}(\text{nil}, k) = k\text{nil}$

$\text{rev}(\text{cons}(d, h, t), k) = \text{rev}(t, \lambda l. k(\text{append}(t, \text{cons}(d, h, \text{nil}))))$

Idea: $\text{rev}(l, k) = k(\text{reverse}(l))$.

Higher-order functions and size

With

$$|f|_{A \rightarrow B} = \sup\{|f(a)|_B - |a|_A \mid a \in \llbracket A \rrbracket\}$$

functions continue to be non size increasing.

However, storing f takes more space than $|f|_{A \rightarrow B}$.

Structural recursion with HO functions

Theorem: (MH1999, Aehlig-Schwichtenberg 2000, Dal Lago & MH 2005):

The functions definable with structural recursion and HO functions are polynomial time computable.

Proof idea: semantic interpretation of programs (also higher-order ones) as time and space bounded algorithms. AS use normalisation.

Let us call that system (structural recursion formalised e.g. with iterators) $LFPL_{\text{poly}}$

Resource monoid

Define M as the set of pairs (l, p) where $l \in \mathbb{N}$ and p is a unary polynomial.

Define a binary addition on M by $(l_1, p_1) + (l_2, p_2) = (l_1 + l_2, p_1 + p_2)$.

Define a partial ordering on M by $(l_1, p_1) \leq (l_2, p_2)$ if $l_1 \leq l_2$ and furthermore $p_1(x) \leq p_2(x)$ for all $x \geq l_2$.

$(M, +)$ is a monoid and $+$ is monotone w.r.t. \leq .

If $\alpha_1 = (l_1, p_1)$ and $\alpha_2 = (l_2, p_2)$ and $\alpha_1 \leq \alpha_2$ define $dist(\alpha_1, \alpha_2) = p_2(l_2) - p_1(l_2)$. Note that $dist(\alpha_1, \alpha_2) \in \mathbb{N}$.

The elements of the form $(0, p)$ form a submonoid M_0 of M .

LFPL space

An LFPL-space A comprises

- A set $|A|$ or A for short
- A relation $\mathbf{maj}_A \subseteq \mathit{Machines} \times M \times A$ such that:
 - for each $a \in |A|$ there exists t, α such that $t, \alpha \mathbf{maj}_A a$.
 - if $t, \alpha \mathbf{maj}_A a$ and $\beta \geq \alpha$ then $t, \beta \mathbf{maj}_A a$.
 - $e, \alpha \Vdash_A a$ implies $\mathit{dist}(0, \alpha) \geq |e|$ where $|e|$ is the size of machine e and 0 is $(0, 0) \in M$.
 - $t, \alpha_1 \mathbf{maj}_A a_1$ and $t, \alpha_2 \mathbf{maj}_A a_2$ implies $a_1 = a_2$ so that elements are uniquely characterised by their realisers.

Examples of LFPL-spaces

- $|B| = \{\mathbf{tt}, \mathbf{ff}\}$, $t, \alpha \mathbf{maj}_B x$ when t encodes x and $\alpha = (l, p)$ where $\mathit{dist}(0, \alpha) \geq |t|$.
- $|I| = \{0\}$, $t, \alpha \mathbf{maj}_I 0$ iff $\mathit{dist}(0, \alpha) \geq |t|$.
- $|L(B)| = \{\text{lists of Booleans}\}$ and $t, \alpha \mathbf{maj}_A a$ iff t encodes a and $\alpha = (l, p)$ where $l \geq \text{lh}(a)$ and $\mathit{dist}(0, \alpha) \geq |t|$.
- $|\diamond| = \{0\}$, $t, \alpha \mathbf{maj}_\diamond 0$ iff $\alpha \geq (1, 0)$ and $\mathit{dist}(0, \alpha) \geq |t|$.

LFPL-spaces as a category

A map from LFPL-space A to B is a function $f : |A| \rightarrow |B|$ such that

- there exists $e \in \mathit{Machines}$ and $\phi \in M_0$ (not $M!!$). If we only find a $\phi \in M$ then f is an M -map, not a map.
- whenever $t, \alpha \mathbf{maj}_A a$ then $\{e\}(t), \beta \mathbf{maj}_B f(b)$ for some $\beta \leq \phi + \alpha$ and, moreover, $\mathit{Time}(\{e\}(t)) \leq \mathit{dist}(\beta, \phi + \alpha)$.

Examples of maps

The maps from $L(B)$ to $L(B)$ are precisely the non size-increasing polytime functions.

Suppose that $f : |L(B)| \rightarrow |L(B)|$ satisfies $\text{lh}(f(a)) \leq \text{lh}(a)$ and $f(a)$ is computable in time $p(\text{lh}(a))$. Then f is a map from $L(B)$ to $L(B)$ by $(0, p)$ as follows: Suppose that e is an algorithm for f running in time p and put $\phi = (0, p)$. Suppose that $t, \alpha \mathbf{maj}_{L(B)} a$. We may assume $t = (\text{lh}(a), q)$ where $|t| \leq q(\text{lh}(a))$.

Now $\{e\}(t), \alpha \mathbf{maj}_{L(B)}$ and $\text{dist}(\alpha, \alpha + \phi) = p(\text{lh}(a))$ which equals the runtime of $\{e\}(t)$.

The converse is clear.

LFPL-spaces as a model of $LFPL_{\text{poly}}$

LFPL-spaces form a symmetric monoidal closed category (thus model \otimes, \multimap) irrespective of the choice of the resource monoid.

To wit, $|A \otimes B| = |A| \times |B|$ and $\langle s, t \rangle, \gamma \mathbf{maj}_{A \otimes B} (a, b)$ when $\gamma \geq \alpha + \beta$ for some α, β such that $s, \alpha \mathbf{maj}_A a$ and $t, \beta \mathbf{maj}_B b$ and also $\text{dist}(0, \gamma) \geq |\langle s, t \rangle|$.

We require here that $|\langle s, t \rangle| = |s| + |t| + O(1)$.

Linear function space

We put $|A \multimap B|$ the set of all M -maps from A to B and

$e, \phi \mathbf{maj}_{A \multimap B} f$ if e, ϕ witness that f is an M -map and furthermore $dist(0, \phi) \geq |e|$.

In this way, LFPL-spaces become a symmetric monoidal closed category (model of multiplicative linear logic, linear lambda calculus) and this works for any resource monoid (obvious abstraction from M, M_0).

An LFPL-map from I to A is the same as an element a of $|A|$ such that there exists t and $\alpha \in M_0$ with $t, \alpha \mathbf{maj}_A a$.

An LFPL-map from I to $A \multimap B$ is the same as an LFPL-map from A to B .

Recursion Theorem

Proposition: $\text{cons} : \diamond \multimap B \multimap L(B) \multimap L(B)$ and $\text{nil} : I \multimap L(B)$ are maps.

Theorem: When X is any LFPL-space, and $f : \diamond \multimap B \multimap X \multimap X$ then $\text{rec}(f)$ defined by $\text{rec}(f)(\text{nil})(x) = x$ and $\text{rec}(f)(\text{cons}(d, b, a))(x) = f(d)(b)(a)(\text{rec}(f)(x))$ is an LFPL map of type $L(B) \multimap X \multimap X$.

We will deduce this from a more general result.

Powering

Let A be an LFPL-space and $n \in \mathbb{N}$. We form a new LFPL-space $A^{(n)}$ by $|A^{(n)}| = |A|$ and $t, \alpha \mathbf{maj}_{A^{(n)}} a$ if there exists α' such that $t, \alpha' \mathbf{maj}_A a$ and $\alpha \geq \alpha' + \dots + \alpha'$ (n summands).

If A_i is a family of LFPL-spaces indexed by $i \in I$, e.g., $I = \mathbb{N}$ or $I = \{\text{LFPL-spaces}\}$.

We form the LFPL-space $\forall i.A_i$ with $|\forall i.A_n| = \prod_{i \in I} A_i$ and $t, \alpha \mathbf{maj}_{\forall n.A_n} a$ if $t, \alpha \mathbf{maj}_{A_i} a_i$ for all $i \in I$.

Likewise, $\exists i.A_i$ with $|\exists i.A_n| = \sum_{i \in I} A_i$ and $t, \alpha \mathbf{maj}_{\exists n.A_n} (i, a)$ if $t, \alpha \mathbf{maj}_{A_i} a$. For cardinality reasons we have $\exists i.A_i$ only for $I = \mathbb{N}$.

Examples of LFPL-maps:

$$\forall m. \forall n. A^{(m+n)} \rightarrow A^{(m)} \otimes A^{(n)}$$

$$\forall X. \forall Y. \forall Z. (X \otimes Y \multimap Z) \multimap (X \multimap Y \multimap Z)$$

Finally: The LFPL-bang

If A is an LFPL-space we form a new LFPL-space $!A$ with $!|A| = |A|$ and $t, \alpha \mathbf{maj}_{!A} a$ if there exists $\alpha_0 = (0, p) \in M_0$ such that $t, \alpha_0 \mathbf{maj}_A a$ and $\alpha \geq (0, (1+x)p)$.

“!” is a functor and the following obvious functions are LFPL-maps:

$$!(A \otimes B) \leftrightarrow !A \otimes !B$$

$$\forall n. !A \otimes \diamond^{(n)} \multimap A^{(n)} \otimes \diamond^{(n)}$$

This allows us to *define*

$$L(B) = \exists n. \diamond^{(n)} \otimes \forall X. X \multimap (B \multimap X \multimap X)^{(n)} \multimap X$$

and similarly for other datatypes and recover constructors and iterators, e.g.,

$$rec :!(\diamond \multimap B \multimap X \multimap X) \multimap L(B) \multimap X \multimap X$$

HO functions & general recursion

$\text{Contrived} : A, A \multimap A \rightarrow A$

$e_{\text{Contrived}}(x, f) = \text{match } a(x) \text{ with } x' \otimes b \otimes \text{done} \Rightarrow \text{if } \text{done} \text{ then } f(x')$

 else if b then $\text{Contrived}(x', \lambda y. g(f(y)))$

 else $\text{Contrived}(x', \lambda y. f(h(y)))$

where a, g, h are arbitrary function symbols of appropriate types.

Evaluating $\text{Contrived}(\lambda y. y, x)$ by term rewriting may require an arbitrary amount of space.

Is general recursion with higher-order linear functions Turing complete?

Do higher-order linear functions buy any extra expressive power beyond first-order functions?

Higher-order functions represent polynomial space

Theorem: Every function $f : \{0, 1\}^* \rightarrow \{0, 1\}$ computable in polynomial space with stack (equivalently *EXPTIME* can be represented in $LFPL_\omega$.) □

An element $w : L(\mathbb{B}^{\otimes c})$ can store $c|w|$ bits. For p a polynomial we need a type S_p such that an element $w : S_p$ can store $p(|w|)$ bits.

If we had such a type S_p we could use it to store configurations of a p -space bounded TM and use general recursion as before to represent stack.

What does it mean to “store x bits”?

What should the type S_p be?

Notation

$$W = \llbracket L(B) \rrbracket$$

$$W_k = \{w \in W \mid |w| = k\}$$

We identify W_k with integers $< 2^k$. E.g. $W_5 \ni [\text{ff}, \text{tt}, \text{tt}, \text{ff}, \text{ff}] = 12$.

Using higher-order functions as store

Proposition: For any type S there is a program containing a symbol

$$\Phi : L(S) \longrightarrow (L(B) \multimap L(S)) \otimes L(B)$$

such that $\llbracket \Phi \rrbracket(l) = (f, w)$ implies

- $\text{lh}(l) = \text{lh}(w)$,
- whenever $\text{lh}(w') = \text{lh}(w)$ then $f(w') = l$

Thus, if S has size 0, we can store the information contained in a size n object (l) in a size 0 object (f). To retrieve the information we need to provide an arbitrary size n object (w') as a “catalyst”.

Storage device

Let $s : \mathbb{N} \rightarrow \mathbb{N}$ be a function with $s(n) < 2^n$ and $k \in \mathbb{N}$ be a number. A (k, s) -storage device is given by the following data:

- a type S
- a family of subsets $S_n \subseteq \llbracket S \rrbracket$ for $n \in \mathbb{N}$.
- a represented function $\text{init} : \rightarrow S$,
- a represented function
 $\text{read} : L(B) \otimes L(B) \otimes S \rightarrow L(B) \otimes L(B) \otimes B \otimes S$
- a represented function
 $\text{write} : L(B) \otimes L(B) \otimes B \otimes S \rightarrow L(B) \otimes L(B) \otimes S$.

such that, *please turn*

For all $n \in \mathbb{N}$ and $w, w_1, w_2, w_3 \in W_{kn}$, $a, a' \in W_n$ and $s \in S_n$ the following are satisfied:

- $\llbracket \text{init} \rrbracket () \in S_n$
- $\llbracket \text{read} \rrbracket (w, a, s) = (w', a', b, s')$ implies $w' \in W_{kn}, a' \in W_n, s' = s$
- $\llbracket \text{write} \rrbracket (w, a, b, s) = (w', a', s')$ implies $w' \in W_{kn}, a' \in W_n, s' \in S_n$
- $\llbracket \text{read} \rrbracket (w_1, a, \llbracket \text{write} \rrbracket (w_2, a, b, s).2.2).2.2.1 = b$ provided that $a < s(n)$
- $\llbracket \text{read} \rrbracket (w_1, a, \llbracket \text{write} \rrbracket (w_2, a', b, s).2.2) = \llbracket \text{read} \rrbracket (w_3, a, s)$ provided that $a, a' < s(n)$ and $a \neq a'$.

Building storage devices

Fact: $S := B^{\otimes c}$ can be extended to a $(0, \lambda n.c)$ -storage device.

I.e., a constant number of bits is stored, no scratchpad is required.

Lemma: If S is a (k, s) storage device then $S' := L(B) \multimap L(S)$ can be extended to a $(k + 1, \lambda n.n \cdot s(n))$ storage device.

Corollary: If p is a polynomial of degree k then there exists a (k, p) storage device.

Representing polyspace + stack

Let $f : W \rightarrow \{0, 1\}$ be computable in space $p(n)$ plus stack.

We let S be a $(k, \lambda n.p(2kn))$ storage device.

Given input $l : L(B)$ we first split l into $w : L(B)$ and $l' : L(B \otimes B)$ so that $|w| = |l'| = |l|/2$ but l' contains the same information as l .

Using w as scratchpad our storage device can thus store $p(n)$ bits as required.

Upper bound on expressivity by dynamic programming

Theorem: If f is representable with general recursion and HO-functions then $f \in \text{DTIME}(2^{n^{O(1)}})$.

Proof: Intuitions:

- When evaluating $f(w)$ all intermediate first-order results should be bounded (length-wise) by $|w|$, so a finite model should suffice.
- Assuming nondeterminism a linear (apply once) function can be represented by an argument-result pair.

Let $N \in \mathbb{N}$ be a fixed parameter. We define finite sets $\llbracket A \rrbracket$ for types A inductively as follows.

$$\llbracket \diamond \rrbracket = \{0\} \quad \llbracket \mathbf{B} \rrbracket = \{\mathbf{tt}, \mathbf{ff}\}$$

$$\llbracket \mathbf{L}(A) \rrbracket = \{w \in \llbracket A \rrbracket^* \mid |w|_{\mathbf{L}(A)} \leq N\} \quad \llbracket A \multimap B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket$$

$$\llbracket A \otimes B \rrbracket = \{x \in \llbracket A \rrbracket \times \llbracket B \rrbracket \mid |x|_{A \otimes B} \leq N\}$$

$$\llbracket \Gamma \rrbracket = \{\eta \mid \forall x \in \text{dom}(\Gamma). \eta(x) \in \llbracket \Gamma(x) \rrbracket \wedge |\eta|_{\Gamma} \leq N\}$$

If $\Gamma \vdash e : A$ then $\llbracket e \rrbracket \subseteq \llbracket \Gamma \rrbracket \times \llbracket A \rrbracket$. E.g.

$$\llbracket e_1 e_2 \rrbracket(\eta_1, \eta_2) = \{y \mid \exists x. x \in \llbracket e_2 \rrbracket(\eta_2) \wedge (x, y) \in \llbracket e_1 \rrbracket(\eta_1)\}$$

General recursion rendered by (finite) iteration of monotone operator.

If $R \subseteq A \times B$ and $U \subseteq A$ write $R(A)$ for $\{b \mid \exists a \in A. aRb\}$

Main result about $\llbracket - \rrbracket$

Recall that $\llbracket - \rrbracket : \begin{array}{l} \text{Types} \rightarrow \text{cpos} \\ \text{Terms} \rightarrow \text{cts fctns} \end{array}$

In particular $\llbracket A \multimap B \rrbracket = \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket_{\perp}$.

Proposition: Suppose that $f : L(B) \rightarrow L(B)$ in some program and let $l \in W$ where $|l| \leq N$ (recall that N is a fixed parameter). Notice that in this case $l \in \llbracket L(B) \rrbracket$ as well as $l \in \llbracket L(B) \rrbracket$. Then

$$l \llbracket f \rrbracket l' \iff \llbracket f \rrbracket(l) = l'$$

for all $l' \in W$.

This will follow from a more general property proved by induction on typing derivations.

Correspondence between $\llbracket - \rrbracket$ and $\langle - \rangle$

For each type A and $n \in \mathbb{N}$ define relation

$\sim_A^n \subseteq \llbracket A \rrbracket \times \{U \subseteq \langle A \rangle \mid U \neq \emptyset \wedge |U|_A \leq n\}$ where $|U|_A = \max_{x \in U} |x|_A$.

Intention: $x \sim_A^n U$ means U consists of size $\leq n$ elements all encoding x .

Clauses for \sim_A^n

$$\mathbf{tt} \sim_B^n \{\mathbf{tt}\} \quad \mathbf{ff} \sim_B^n \{\mathbf{ff}\} \quad 0 \sim_{\diamond}^{n+1} \{0\} \quad [] \sim_{L(A)}^n \{[]\}$$

$$(a, b) \sim_{A \otimes B}^n W \iff$$

$$\exists n_1, n_2, U, V. n_1 + n_2 = n$$

$$\wedge a \sim_A^{n_1} U \wedge b \sim_B^{n_2} V \wedge W = U \times V$$

Clauses for \sim_A^n

$$f \sim_{A \multimap B}^n U \iff$$

$$\forall n_1, x. n + n_1 \leq N$$

$$\wedge x \sim^{n_1} V \Rightarrow f(x) \sim_B^{n+n_1} U(V)$$

$$x :: l \sim_{L(A)}^n W \iff$$

$$\exists n_1, n_2, U, V. n_1 + n_2 + 1 \leq n$$

$$\wedge x \sim_A^{n_1} U \wedge l \sim_{L(A)}^{n_2} V \wedge W = U :: V$$

Formally extend by $\perp \sim_A^n \emptyset$.

Main Lemma

Suppose given denotations for function symbols $\psi(f) \subseteq \llbracket \vec{A} \rrbracket \rightarrow \llbracket B \rrbracket \perp$ and $\rho f \subseteq \llbracket \vec{A} \rrbracket \times \llbracket B \rrbracket$ such that whenever $n = n_1 + \dots + n_r \leq N$ one has $\bigwedge_i u_i \sim_{A_i}^{n_i} U_i \implies \psi(f)(u_1, \dots, u_r) \sim_B^n \rho(f)^n(U_1, \dots, U_r)$

Lemma: If $\Gamma \vdash e : B$ and $\eta \sim_{\Gamma}^n \vec{U}$ then $\llbracket e \rrbracket_{\psi, \eta} \sim_B^n \llbracket e \rrbracket_{\rho, \vec{U}}$.

Proof by induction on typing derivations.

For a fixed set of types \mathcal{T} , e.g., those occurring in a given program there is a polynomial p such that $\log \#(A) \leq p(N)$ for all $A \in \mathcal{T}$.

A value table for (f) takes space $2^{p(N)}$.

By maintaining space for two value tables for (f) we can compute (f) in time $2^{p(N)}$.

Hence all those functions are computable in time $O(2^{p(N)})$.

To know more See papers on [www.tcs.ifi.lmu.de/ mhofmann](http://www.tcs.ifi.lmu.de/~mhofmann) or citeseer.

Results on $LFPL_\omega$ (General recursion and higher-order linear functions):
POPL02 — The strength of non-size-increasing computation. This paper also shows that all polytimes functions are representable in $LFPL_{\text{poly}}$.

Results on $LFPL_{\text{poly}}$ (Definable functions contained in PTIME, extensions for PSPACE, divide and conquer algorithms etc.): Non size-increasing polynomial time computation (Inf. & Comp. 2001).

New proof using resource monoids: with Ugo Dal Lago. Quantitative models and Implicit Complexity. FSTTCS 05. Full version on Ugo's home page. Also contains soundness proofs for LLL, ELL, SLL.

Inference of Diamond types: Static Prediction of Heap Space Usage for First-Order Functional Programs. POPL03. With Steffen Jost.