

# Functional Programming in Sublinear Space

Ulrich Schöpp

Institute of Advanced Studies  
University of Bologna  
(on leave from University of Munich)

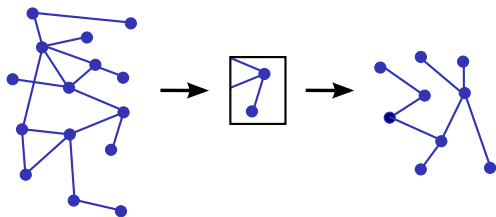
Joint work with Ugo Dal Lago

Workshop on Geometry of Interaction, Traced Monoidal Categories  
and Implicit Complexity, Kyoto, August 2009

# Programming with Sublinear Space

## Computation with data that does not fit in memory

- Input can be requested in small chunks from the environment.
- Output is provided piece-by-piece.



## Writing sublinear space programs can be quite complicated

- Cannot store intermediate values.
- Recompute small parts of values only when they are needed.

# Language/Compiler Support

Can we find a Programming Language that

- hides on-demand recomputation behind useful abstractions;
- delegates some tedious programming tasks to a compiler;
- allows for an easy combination of a sublinear space algorithms with the rest of the program?

# Language/Compiler Support

Existing work for LOGSPACE explores possible abstractions ...

- restricted primitive recursion [Møeller-Neergaard 2004]
- subsystem of Bounded Linear Logic [Sch. 2007]
- (LOGSPACE predicates: [Kristiansen 2005], [Bonfante 2006])

...but is still far away from making programming easier.

Perhaps it is too ambitious for now to aim for a programming language that completely hides on-demand recomputation?

A more modest goal:

Design a functional programming language that provides support for working with sublinear space, without trying to hide all implementation details behind abstractions.

# A Functional Language for Sublinear Space

1. **Computation with external data**

How should we represent data that does not fit into memory in a functional programming language?

2. Deriving the functional language IntML

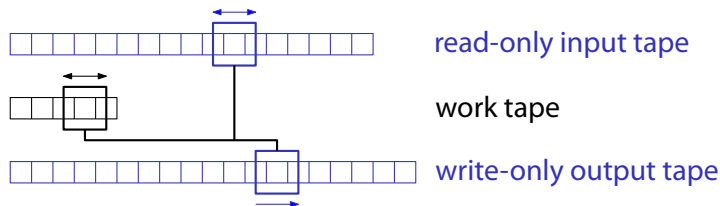
3. LOGSPACE programming in IntML

# Sublinear Space Complexity

Modify the machine model to account for computation with external data:

Turing Machines  $\Rightarrow$  Offline Turing Machines

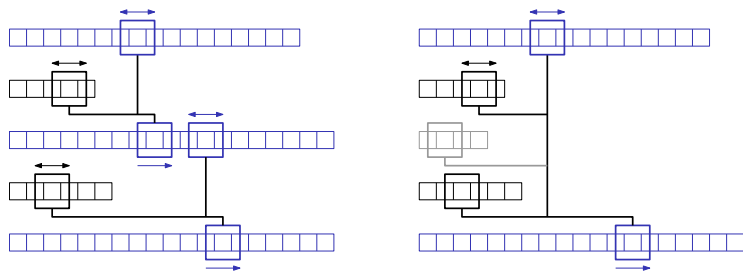
## Offline Turing Machines



Input and output tape belong to environment.  
Only the space on the work tape(s) counts.

# Offline Turing Machines

Composition is implemented without storing intermediate result.



# Offline Turing Machines

Offline Turing Machines can be seen as a convenient abbreviation for normal Turing Machines that

- obtains its input not in one piece but that may request it character-by-character from the environment;
- gives its output as a stream of characters.

Formally, we may describe this as a computable function of type

$$\mathbb{N} + (\textit{State} \times \Sigma) \longrightarrow \Sigma + (\textit{State} \times \mathbb{N}) .$$

Request for output character

Output character

Request for input character

Answer for input request



# Space Complexity in Functional Programs

What relates to Offline Turing Machines in the same way that functional programming languages relate to Turing Machines?

Turing Machines ————— Functional Languages  
(OCaml, Haskell, ...)

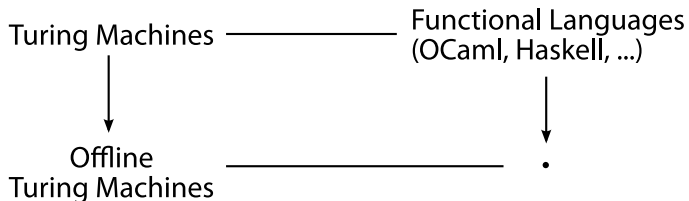
Offline Turing Machines ————— ?

# Int Construction

Understand the transition from Turing Machines to Offline Turing Machines in terms of the *Int construction*

[Joyal, Street & Verity 1996].

1. Apply the Int construction directly to a functional language.



2. Derive a functional language from the semantic structure.
3. Identify programs with sublinear space usage.

# Traced Monoidal Category

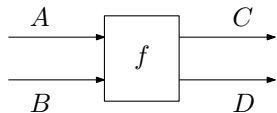
- Category  $\mathbb{B}$

(e.g. sets and partial functions)

- Monoidal structure  $(+, 0)$

(e.g. disjoint union)

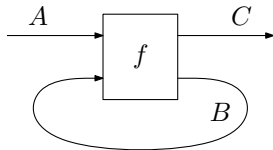
$$f: A + B \longrightarrow C + D$$



- Trace

(e.g. while loop)

$$\frac{f: A + B \longrightarrow C + B}{\text{Tr}(f): A \longrightarrow C}$$

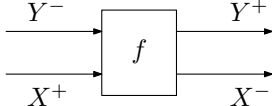


# Category $\text{Int}(\mathbb{B})$

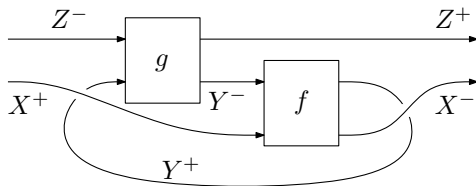
- Objects are pairs of  $\mathbb{B}$ -objects

$$X = (X^-, X^+)$$

- Morphism  $f: X \rightarrow Y$  is a  $\mathbb{B}$ -map

$$f: X^+ + Y^- \longrightarrow X^- + Y^+$$


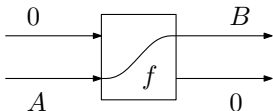
- Composition



# Structure in $\text{Int}(\mathbb{B})$

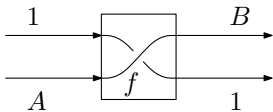
## $\mathbb{B}$ embeds into $\text{Int}(\mathbb{B})$

- A map from  $A \rightarrow B$  in  $\mathbb{B}$  corresponds to a map  $(0, A) \rightarrow (0, B)$  in  $\text{Int}(\mathbb{B})$ .



This gives a full and faithful embedding.

- We shall use  $[A] = (1, A)$ , where  $1$  is a singleton.



# Structure in $\text{Int}(\mathbb{B})$

$\text{Int}(\mathbb{B})$  has a monoidal structure  $\otimes$

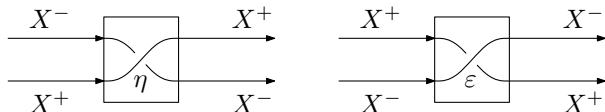
$$(A \otimes B)^- = A^- + B^- \qquad I = (0, 0)$$

$$(A \otimes B)^+ = A^+ + B^+$$

$\text{Int}(\mathbb{B})$  is compact closed

$$(X^-, X^+)^* = (X^+, X^-)$$

Unit  $\eta: I \rightarrow X^* \otimes X$  and counit  $\varepsilon: X \otimes X^* \rightarrow I$ :



$\mathbb{B}$  is monoidal closed  $X \multimap Y = X^* \otimes Y$

# Structure in $\text{Int}(\mathbb{B})$

$\text{Int}(\mathbb{B})$  has  $\mathbb{B}$ -object indexed tensors

$$\left(\bigotimes_A X\right)^- = A \times X^-$$
$$\left(\bigotimes_A X\right)^+ = A \times X^+$$

(given suitable structure in  $\mathbb{B}$ , e.g. products)

## Example

$\mathbb{B}$  sets with partial functions,  $(+, 0)$  coproduct,  $A$  finite

$$\bigotimes_A X \cong \underbrace{X \otimes \dots \otimes X}_{|A| \text{ times}}$$

# Indexed Tensor

Consider the  $\text{Int}$ -construction on categories  $\mathbb{B}$  with

- finite products  $(\times, 1)$ ;
- distributive finite coproducts  $(+, 0)$ ;
- uniform trace with respect to  $(+, 0)$ .

Define  $\mathbb{B}[\Sigma]$  by freely adjoining to  $\mathbb{B}$  an indeterminate element of  $\Sigma$ .

One obtains indexed categories:

$$\begin{aligned}\mathbb{B}[-] &: \mathbb{B}^{\text{op}} \rightarrow \text{Cat} \\ \text{Int}(\mathbb{B}[-]) &: \mathbb{B}^{\text{op}} \rightarrow \text{Cat}\end{aligned}$$

The indexed tensor is a strong monoidal functor

$$\bigotimes_A : \text{Int}(\mathbb{B}[\Sigma \times A]) \rightarrow \text{Int}(\mathbb{B}[\Sigma]).$$



# Indexed Tensor

$$\bigotimes_A : \text{Int}(\mathbb{B}[\Sigma \times A]) \rightarrow \text{Int}(\mathbb{B}[\Sigma])$$

- For any  $f: \Sigma \rightarrow A$  a strong monoidal natural transformation:

$$\pi_f: \bigotimes_A X \longrightarrow \langle \Sigma, f \rangle^* X \quad \text{in } \text{Int}(\mathbb{B}[\Sigma])$$

- Natural isomorphisms that are compatible with  $\pi_f$ .

$$\bigotimes_1 X \cong \rho^* X$$

$$\bigotimes_{A+B} X \cong \left( \bigotimes_A (\Sigma \times \text{inl})^* X \right) \otimes \left( \bigotimes_B (\Sigma \times \text{inr})^* X \right)$$

$$\bigotimes_{A \times B} X \cong \bigotimes_A \bigotimes_B \alpha^* X$$

# Indexed Tensor

The projections

$$\pi_f: \bigotimes_A X \longrightarrow \langle \Sigma, f \rangle^* X \quad \text{in } \text{Int}(\mathbb{B}[\Sigma])$$

internalise to a certain extent:

$$\begin{array}{ccc} \bigotimes_A [B] & \xrightarrow{[f] \otimes id} & [A] \otimes \bigotimes_A [B] \\ & \searrow \pi_f & \downarrow \pi \\ & & [B] \end{array} \quad \text{in } \text{Int}(\mathbb{B}[\Sigma])$$

# Int Construction and Space Complexity

The functions that represent Offline Turing Machines

$$(State \times \Sigma) + \mathbb{N} \longrightarrow (State \times \mathbb{N}) + \Sigma$$

appear in  $\text{Int}(\mathbf{Pfn})$  as morphisms of type

$$\bigotimes_{State} (\mathbb{N} \multimap \Sigma) \longrightarrow (\mathbb{N} \multimap \Sigma),$$

where we write just  $\mathbb{N}$  for  $(0, \mathbb{N})$  and  $\Sigma$  for  $(0, \Sigma)$ .

# Int Construction and Space Complexity

The functions that represent Offline Turing Machines

$$(State \times \Sigma) + \mathbb{N} \longrightarrow (State \times \mathbb{N}) + \Sigma$$

appear in  $\text{Int}(\mathbf{Pfn})$  as morphisms of type

$$\bigotimes_{State} (\mathbb{N} \multimap \Sigma) \longrightarrow (\mathbb{N} \multimap \Sigma),$$

where we write just  $\mathbb{N}$  for  $(0, \mathbb{N})$  and  $\Sigma$  for  $(0, \Sigma)$ .

The structure of  $\text{Int}(\mathbf{Pfn})$  is useful for working with OTMs.

- Composition:

$$\bigotimes_S \bigotimes_{S'} (\mathbb{N} \multimap \Sigma) \xrightarrow{\otimes_{S'} f} \bigotimes_S (\mathbb{N} \multimap \Sigma) \xrightarrow{g} (\mathbb{N} \multimap \Sigma)$$

- Input lookup is just (linear) function application.
- ...

# A Functional Language for Sublinear Space

1. Computation with external data
2. Deriving the functional language IntML
  1. Start with a standard functional programming language.
  2. Apply the Int construction to a *term model*  $\mathbb{B}$  of this language.
  3. Derive a functional language from the structure of  $\text{Int}(\mathbb{B})$ . It can be seen as a *definitional extension* of the initial language.
  4. Identify programs with sublinear space usage.
3. LOGSPACE programming in IntML

# A Simple First Order Language

## Finite Types

$$A, B ::= \alpha \mid A + B \mid 1 \mid A \times B$$

## Ordering on all types

$$\text{min}_A \mid \text{succ}_A(f) \mid \text{eq}_A(f, f)$$

## Explicit trace (with respect to +)

$$\text{trace}(c.f)(g)$$

(sufficient for now, could use tail recursion)

Standard call-by-value evaluation, constants unfolded on demand

Chosen for simplicity and to make analysis easy.  
Richer languages are possible.

# Examples

## Example: Addition

$$x: \alpha, y: \alpha \vdash \text{add}(x, y): \alpha$$

## With syntactic sugar for tail recursion:

```
add(x, y) =  
  if y = min then x else add(succ x, pred y)
```

## With explicit trace:

```
add(x, y) =  
  (trace p. case p of  
    inl(z) -> inr(z)  
    | inr(z) -> let z be <x, y> in  
      if y = min then inl(x) else inr(<succ x, pred y>)  
  ) <x,y>
```

# The Functional Language IntML

IntML extends the simple first order language with syntax for  $\text{Int}(\mathbb{B})$ , where  $\mathbb{B}$  is the term model of the simple first order language.

IntML has two classes of terms and types:

- Working Class (for  $\mathbb{B}$ )

$$A, B ::= \alpha \mid A + B \mid 1 \mid A \times B$$

Terms from the simple first order language + unbox

- Upper Class (for  $\text{Int}(\mathbb{B})$ )

$$X, Y ::= [A] \mid X \otimes Y \mid A \cdot X \multimap Y$$

All computation is being done by working class terms. Upper class terms correspond to morphisms in  $\text{Int}(\mathbb{B})$ , which are implemented by working class terms.



# IntML Type System — Working Class

Usual typing rules, e.g.

$$\frac{\Sigma \vdash f : A \quad \Sigma \vdash g : B}{\Sigma \vdash \langle f, g \rangle : A \times B}$$

...

There is one additional rule for using upper class results in the working class:

$$\frac{\Sigma \mid \vdash t : [A]}{\Sigma \vdash \text{unbox } t : A}$$

# IntML Type System — Upper Class

The upper class type system identifies a useful part of  $\text{Int}(\mathbb{B})$ .

## Types

$$X, Y ::= [A] \mid X \otimes Y \mid A \cdot X \multimap Y$$

In the syntax we write  $A \cdot X$  for  $\bigotimes_A X$ .

## Typing Sequents

$$\Sigma \mid x_1 : A_1 \cdot X_1, \dots, x_n : A_n \cdot X_n \vdash t : Y$$

denotes morphism

$$\bigotimes_{A_1} X_1 \otimes \dots \otimes \bigotimes_{A_n} X_n \longrightarrow Y \quad \text{in } \text{Int}(\mathbb{B}[\Sigma]).$$

The restrictions on the appearance of  $\bigotimes_A$  are motivated by Dual Light Affine Logic [\[Baillot & Terui 2003\]](#).

# Upper Class Typing Rules

$$\text{(Var)} \frac{}{\Sigma \mid \Gamma, x : A \cdot X \vdash x : X}$$

$$\text{(LocWeak)} \frac{\Sigma \mid \Gamma, x : A \cdot X \vdash s : Y}{\Sigma \mid \Gamma, x : (B \times A) \cdot X \vdash s : Y}$$

$$\text{(Congr)} \frac{\Sigma \mid \Gamma, x : A \cdot X \vdash s : X}{\Sigma \mid \Gamma, x : B \cdot X \vdash s : X} \quad A \cong B, \text{ e.g. } 1 \times A \cong A$$

$$\text{(-}\circ\text{-I)} \frac{\Sigma \mid \Gamma, x : A \cdot X \vdash s : Y}{\Sigma \mid \Gamma \vdash \lambda x. s : A \cdot X \multimap Y}$$

$$\text{(-}\circ\text{-E)} \frac{\Sigma \mid \Gamma \vdash s : A \cdot X \multimap Y \quad \Sigma \mid \Delta \vdash t : X}{\Sigma \mid \Gamma, A \cdot \Delta \vdash s t : Y}$$

(straightforward rules for  $\otimes$ )

# Upper Class Typing Rules

$$\text{(Contr)} \frac{\Sigma \mid \Gamma \vdash s : X \quad \Sigma \mid \Delta, x : A \cdot X, y : B \cdot X \vdash t : Y}{\Sigma \mid \Delta, (A + B) \cdot \Gamma \vdash \text{copy } s \text{ as } x, y \text{ in } t : Y}$$

$$\text{(Case)} \frac{\Sigma \vdash f : A + B \quad \Sigma, c : A \mid \Gamma \vdash s : X \quad \Sigma, d : B \mid \Gamma \vdash t : X}{\Sigma \mid \Gamma \vdash \text{case } f \text{ of } \text{inl}(c) \Rightarrow s \mid \text{inr}(d) \Rightarrow t : X}$$

$$([\ ]\text{-I}) \frac{\Sigma \vdash f : A}{\Sigma \mid \Gamma \vdash [f] : [A]}$$

$$([\ ]\text{-E}) \frac{\Sigma \mid \Gamma \vdash s : [A] \quad \Sigma, c : A \mid \Delta \vdash t : [B]}{\Sigma \mid \Gamma, A \cdot \Delta \vdash \text{let } s \text{ be } [c] \text{ in } t : [B]}$$

## Upper Class Typing — Examples

$$\begin{aligned} \lambda x. \text{ let } x \text{ be } [c] \text{ in } [\langle c, c \rangle] \\ : \alpha \cdot [\alpha] \multimap [\alpha \times \alpha] \end{aligned}$$

$$\begin{aligned} \lambda f. \lambda x. \text{ let } x \text{ be } [c] \text{ in } f [c] [c] \\ : \alpha \cdot ([\alpha] \multimap [\alpha] \multimap [\beta]) \multimap [\alpha] \multimap [\beta] \end{aligned}$$

$$\begin{aligned} \lambda y. \text{ copy } y \text{ as } y_1, y_2 \text{ in} \\ \quad \langle \text{let } y_1 \text{ be } [c] \text{ in } [\pi_1 c], \text{ let } y_2 \text{ be } [c] \text{ in } [\pi_2 c] \rangle \\ : (\gamma + \delta) \cdot [\alpha \times \beta] \multimap [\alpha] \otimes [\beta] \end{aligned}$$

Terms do not contain type annotations.

Conjecture: Inference of most general types is possible.

(have an implementation for the type system without rule (Cong);  
unification up to congruence is decidable).

# Hacking

Have we captured all the structure of  $\text{Int}(\mathbb{B})$ ?

# Hacking

Have we captured all the structure of  $\text{Int}(\mathbb{B})$ ?

No!  $\text{Int}(\mathbb{B})$  has a lot more structure!

Can we ever capture all the useful structure?

# Hacking

Have we captured all the structure of  $\text{Int}(\mathbb{B})$ ?

No!  $\text{Int}(\mathbb{B})$  has a lot more structure!

Can we ever capture all the useful structure?

Let the programmer define the structure he needs himself!

$$\text{(Hack)} \frac{\Sigma, x: X^- \vdash a: X^+}{\Sigma \mid \Gamma \vdash \text{hack } x.a: X}$$

$$[A]^- = 1$$

$$[A]^+ = A$$

$$(X \otimes Y)^- = X^- + Y^-$$

$$(X \otimes Y)^+ = X^+ + Y^+$$

$$(A \cdot X \multimap Y)^- = A \times X^+ + Y^- \quad (A \cdot X \multimap Y)^+ = A \times X^- + Y^+$$

Complexity results remain true in presence of (Hack).



# Hacking

## Example: loop

$$\text{loop}: \alpha \cdot (\beta \cdot [\alpha] \multimap [\alpha + \alpha]) \multimap [\alpha] \multimap [\alpha]$$

$$\text{loop } f \ x_0 = \begin{cases} \text{loop } f \ [y] & \text{if } f \ x_0 \text{ is } [\text{inl}(y)] \\ [z] & \text{if } f \ x_0 \text{ is } [\text{inr}(z)] \end{cases}$$

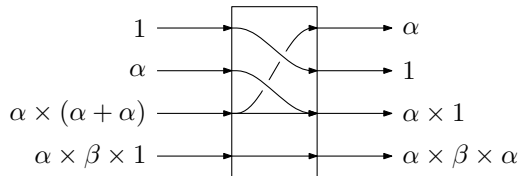
```
loop = hack x. case x of
  | inl(y) -> let y be <store, stepq> in
    case stepq of
      | inl(argq) -> let argq be <argstore, unit> in
        inl(<store, inl(<argstore, store>>>))
      | inr(contOrStop) -> case contOrStop of
          | inl(cont) -> inl(<cont, inr(<>>>))
          | inr(stop) -> inr(inr(stop))
  | inr(z) -> case z of
      | inl(basea) -> let basea be <junk, basea> in
        inl(<basea, inr(<>>>))
      | inr(initialq) -> inr(inl(<<>, <>>>))
```

# Hacking

## Example: loop

$$\text{loop}: \alpha \cdot (\beta \cdot [\alpha] \multimap [\alpha + \alpha]) \multimap [\alpha] \multimap [\alpha]$$

$$\text{loop } f x_0 = \begin{cases} \text{loop } f [y] & \text{if } f x_0 \text{ is } [inl(y)] \\ [z] & \text{if } f x_0 \text{ is } [inr(z)] \end{cases}$$



# A Functional Language for Sublinear Space

1. Computation with external data
2. Deriving the functional language IntML
3. LOGSPACE programming in IntML  
IntML is sound and complete for LOGSPACE.

# LOGSPACE Soundness

Size bounds on working class terms follow easily from the types.

- Types bound the maximal size of closed values.
- We can read off directly from a well-typed term how big its reducts under closed reductions can get, e.g.

$$\begin{aligned}\|c\| &= |A| \quad \text{if } c \text{ is variable of type } A \\ \|\langle f, g \rangle\| &= \|f\| + \|g\| + 1 \\ \|\text{succ}_A(f)\| &= 12|A| + \|f\| \\ &\dots\end{aligned}$$

Type variables as parameters influence the space usage linearly

Let  $x: A \vdash f: B$ , where  $A$  and  $B$  may contain the type variable  $\alpha$ . Then there exist  $m$  and  $n$  such that for any closed type  $C$  and any closed value  $v$  of type  $A[C/\alpha]$

$$f[C/\alpha][v/x] \longrightarrow^* g \quad \text{implies} \quad |g| \leq m \cdot |C| + n.$$

# LOGSPACE Soundness

An upper class term

$$t: A \cdot (B \cdot [\alpha] \multimap [3]) \multimap (C \cdot [P(\alpha)] \multimap [3])$$

represents a LOGSPACE function on binary words.

If we consider  $\alpha$  as a natural number then  $t$  induces a function

$$\varphi_\alpha: \{0, 1\}^{\leq \alpha} \longrightarrow \{0, 1\}^{\leq P(\alpha)}.$$

as follows:

- A word  $w \in \{0, 1\}^{\leq \alpha}$  can be represented as a function in  $\langle w \rangle: B \cdot [\alpha] \multimap [3]$  by a big case distinction.
- Then  $\varphi_\alpha(w)$  is the word that  $(t \langle w \rangle)$  represents.

The working-class term for  $t$  gives a LOGSPACE algorithm for the function

$$w \longmapsto \varphi_{|w|}(w) : \{0, 1\}^* \longrightarrow \{0, 1\}^*.$$

# LOGSPACE Soundness

The compilation of  $t$  is a working-class term  $t'$  of type

$$\begin{aligned} & A \times (B \times 1 + 3) + (C \times P(\alpha) + 1) \\ & \longrightarrow \\ & A \times (B \times \alpha + 1) + (C \times 1 + 3). \end{aligned}$$

It can be understood as an Offline Turing Machine.

Given an input word  $w$ , let  $N = \underbrace{2 \times 2 \times \dots \times 2}_{\lceil \log(|w|) \rceil}$ .

Evaluation of  $s[N/\alpha] v$  needs space linear in  $|N| = \lceil \log(|w|) \rceil$ .

$\Rightarrow$  Can evaluate the output of the Offline Turing Machine represented by  $t'$  in logarithmic space.

# LOGSPACE Completeness

State of a LOGSPACE Turing Machine can be represented as a working class value of type  $S(\alpha)$ .

Step function

$$\text{input}: [\alpha] \multimap [3] \vdash \text{step}: [S(\alpha)] \multimap [S(\alpha) + S(\alpha)]$$

$\text{step} = \lambda x.$  let  $x$  be  $[s]$  in  
let  $\text{input } [inputpos(s)]$  be  $[i]$  in  
[...working class term for transition function ...]

Turing Machine

$$M: A \cdot (B \cdot [\alpha] \multimap [3]) \multimap (C \cdot [P(\alpha)] \multimap [3])$$

$$M = \lambda \text{input}. \lambda \text{outchar}. \text{loop } \text{step } \text{init}$$

# A Functional Language for Sublinear Space

1. Computation with external data
2. Deriving the functional language IntML
3. LOGSPACE programming in IntML

Encoding of Turing Machines is a sanity check at best.

Can the language express LOGSPACE algorithms in a natural way?

- How hard is it to actually write programs?
- Do the types get in the way?
- Can we combine the special sublinear space programming patterns with the standard ones in a useful way?



# Case Studies

Assess the suitability of IntML for LOGSPACE programming by implementing typical algorithms:

- LOGSPACE evaluation of the function algebra  $BC_{\epsilon}^{-}$
- Graph algorithms: deciding acyclicity in undirected graphs

## Function algebra $BC_{\varepsilon}^{-}$ [Møller-Neergaard 2004]

Restricted form of primitive recursion on binary words that captures the functions in LOGSPACE.

- Example basic functions:

$$\text{succ}_0(: y) = y0$$

$$\text{succ}_1(: y) = y1$$

$$\text{case}(: y, t, f) = \begin{cases} t & \text{if } y \text{ ends with } 1 \\ f & \text{otherwise} \end{cases}$$

- Closed under composition
- Closed course-of-value recursion on notation:  
 $f = \text{rec}(g, h_0, h_1, d_0, d_1)$  satisfies

$$f(\vec{x}, \varepsilon : \vec{y}) = g(\vec{x} : \vec{y})$$

$$f(\vec{x}, xi : \vec{y}) = h_i(\vec{x}, x : f(\vec{x}, x \gg |d_i(\vec{x}, x :)| : \vec{y}))$$

# LOGSPACE evaluation of $BC_{\epsilon}^{-}$

Møller-Neergaard proves LOGSPACE soundness by implementing  $BC_{\epsilon}^{-}$  in SML/NJ:

- Binary words are modelled as functions of type  $(\mathbb{N} \rightarrow 3)$ .
- Function  $f(\vec{x}; \vec{y})$  is implemented as SML-function of type

$$(\mathbb{N} \rightarrow 3) \rightarrow \dots \rightarrow (\mathbb{N} \rightarrow 3) \rightarrow (\mathbb{N} \rightarrow 3) .$$

Example:

```
type bit = int option
type input = int -> bit
```

```
fun succ1 (y1 : input) (bt : int) =
  if bt = 0 then SOME 1 else y1 (bt - 1))
```

- Recursion on notation by *computational amnesia*  
[Ong, Mairson, Møller-Neergaard]

# Implementing Recursion by Computational Amnesia

$$g : (\mathbb{N} \rightarrow 3)$$

$$h_0 : (\mathbb{N} \rightarrow 3) \rightarrow (\mathbb{N} \rightarrow 3)$$

$$h_1 : (\mathbb{N} \rightarrow 3) \rightarrow (\mathbb{N} \rightarrow 3)$$

$$f = \text{saferec}(h_0, h_1, g) : (\mathbb{N} \rightarrow 3)$$

$$f(01011) = h_1(h_1(h_0(h_1(h_0(g))))))$$

- Whenever  $h_i$  applies its argument, forget the call stack and just continue.
  - When some  $h_i$  or  $g$  returns a value, we may not know what to do with it — we have forgotten the call stack.
- ⇒ Remember the returned value (one bit) and its depth and restart the computation.

# [Møller-Neergaard 2004]

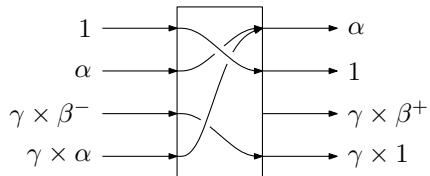
```
exception Restartn
val NORESULT = { depth = ~1, res = NONE, bt=~1 }

fun saferec (g : program-(m-1)-n)
  (h0 : program-m-1) (d0 : program-m-0)
  (h1 : program-m-1) (d1 : program-m-0)
  (x1 : input) ... (xm : input)
  (y1 : input) ... (yn : input) (bt : int) =
  let val result = ref NORESULT
      val goal = ref ({ bt=bt, depth=0 })
      fun loop1 body = if body () then () else loop1 body
      fun loop2 body = if body () then () else loop2 body
      fun findLength (z : input) =
        let fun search i = if z i <> NONE then search (i + 1) else i
            in
              search 0
            end
        fun x' (bt : int) = x1 (1 + bt + #depth (!goal))
        fun recursiveCall (d : program-m-0) (bt : int) =
          let val delta = 1 + findLength (d x' x2 ... xm)
              in
                if #depth (!goal) + delta = #depth (!result)
                    andalso #bt (!result) = bt
                then #res (!result)
                else
                  goal := { bt=bt, depth = #depth (!goal) + delta };
                  raise Restartn
                end
              end
          end
      in
        ( loop1 (fn () => (* Loops until we have the bit at depth 0 *)
          ( goal := { bt=bt, depth=0 };
            loop2 (fn () => (* Loops while the computation is restarted *)
              let val res =
                  case x1 (#depth (!goal)) of
                    NONE => g x2 ... xm y1 ... yn (#bt (!goal))
                  | SOME b =>
                      let val (h, d) = if b=0 then (h0,d0) else (h1,d1)
                          in
                            in
                              h x' x2 ... xm (recursiveCall d) (#bt (!goal))
                            end
                          end
                    in ( result := { depth = #depth (!goal),
                              res = res,
                              bt = #bt (!goal) };
                        true )
                    end handle Restartn => false
                      0 = #depth (!result) ));
              #res (!result))
            end
          end
        end
```

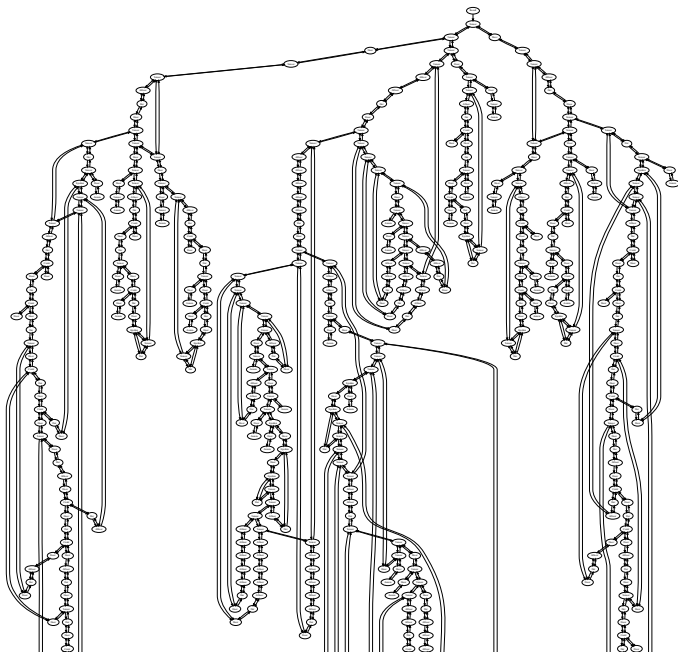
# Control Flow

$\text{callcc} : (\gamma \cdot ([\alpha] \multimap \beta) \multimap [\alpha]) \multimap [\alpha]$

Implemented using hack:



# String Diagram for parity







# Implementing Parity

The term `parity` is just an example to test `saferec`.

The standard algorithm for parity can be implemented easily:

```
parity =U fun x : ['a] --o [1+(1+1)] ->
  let
    loop (fun pos_parityU : ['a * (1+1)]->
      let pos_parityU be [pos_parity] in
      let x [pi1 pos_parity] be [x_pos] in
      case x_pos of
        inl(mblank) -> [inr(pos_parity)]
      | inr(char) ->
          if (pi1 pos_parity) = max then
            [inr(<max, xor char (pi2 pos_parity)>)]
          else
            [inl(<succ (pi1 pos_parity), xor char (pi2 pos_parity)>)]
        ) [<min, false>]
    be [pos_parity] in [pi2 pos_parity];
```

## Further Work — Better Formulation of Control

$$\frac{\Sigma \mid \Gamma \vdash s : \perp \mid \Theta, \alpha : A}{\Sigma \mid \Gamma \vdash \mu\alpha. s : [A] \mid \Theta} \quad \frac{\Sigma \mid \Gamma \vdash s : [A] \mid \Theta, \alpha : A}{\Sigma \mid \Gamma \vdash \text{throw } \alpha s : \perp \mid \Theta, \alpha : A}$$

Example

$$\text{callcc} = \lambda y. \mu\alpha. \text{throw } \alpha (y (\lambda x. \mu\beta. \text{throw } \alpha x))$$

Equations

$$\text{throw } \beta (\mu\alpha. s) = s[\beta/\alpha] : \perp$$

$$\mu\alpha. (\text{throw } \alpha s) = s : [A] \quad \text{if } \alpha \notin FV(s)$$

$$\text{throw } \alpha s = s : \perp$$

$$\text{let } (\mu\beta. \text{throw } \alpha s) \text{ be } [c] \text{ in } t = \mu\gamma. \text{throw } \alpha s \quad \text{if } \beta \neq \alpha$$

## Case Study — Graph Algorithms

Adapt the representation of strings by

$$[\alpha] \multimap [3]$$

to

$$([\alpha] \multimap [2]) \otimes ([\alpha \times \alpha] \multimap [2])$$

for graphs.

Standard pointer program for checking acyclicity in undirected graphs can be implemented without the need for special tricks.

Can implement edge/vertex iterators as higher-order functions.

# Conclusion

Space bounded computation has interesting *structure*, that we have only just begun to explore.

The Int construction seems to be a good first step for capturing that structure precisely.

## Further Work

- Stronger working class calculi: Allowing (certain) function spaces in the working class may lead to polylogarithmic space?
- Completeness: Can we get completeness by something less trivial than `hack`?

