Functional Programming in Sublinear Space

Ulrich Schöpp

University of Munich

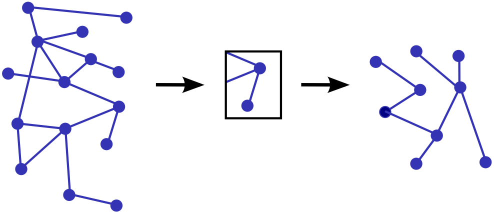
This work was supported by:
Institute of Advanced Studies
University of Bologna

Joint work with Ugo Dal Lago

Programming with Sublinear Space

Computation with data that does not fit in memory

- Input is stored by the environment, allows random access.
- Output is provided piece-by-piece.



Writing such programs can be complicated

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

Language/Compiler Support

How can a programming language support us in writing such algorithms?

Can we find a programming language that

- allows us to forget that certain values do not fit in memory, at least to a certain extent;
- 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 on implicit and logical characterisations of LOGSPACE explores possible abstractions:

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

Supports our goal of finding a functional programming with primitives for sublinear space programming.

Functional Programming With Sublinear Space

1. Computation with external data

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

2. Deriving the functional language IntML

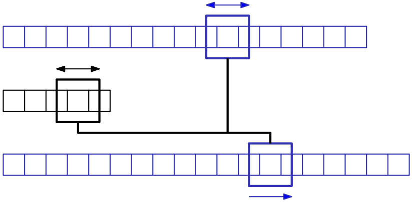
3. Programming in IntML

Sublinear Space Complexity

In complexity theory, one modifies the machine model to account for computation with external data:

Turing Machines $\Longrightarrow$ Offline Turing Machines

Offline Turing Machines



read-only input tape

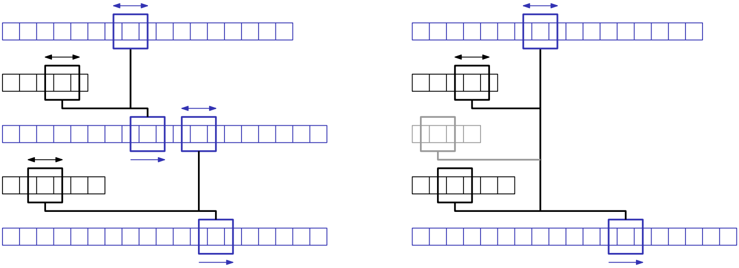
work tape

write-only output tape

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.



⇒ bidirectional data flow

Offline Turing Machines

An Offline Turing Machine can be seen as a convenient abbreviation for a normal Turing Machine 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 that describes how the machine interacts with it environment:

Request for output character

Output character

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

Request for input character

Answer for input request

Functional Programming with External Data

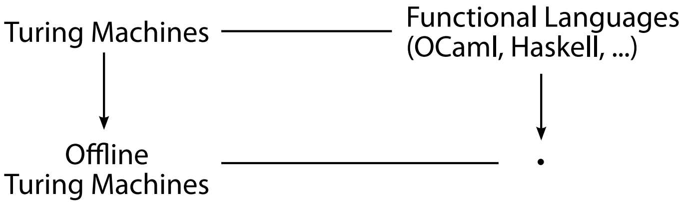
What relates to Offline Turing Machines in the same way that functional programming languages relate to Turing Machines?

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

Offline Turing Machines ———————————— ?

Functional Programming with External Data

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



1. Understand the step from Turing Machines to Offline Turing Machines in terms of the *Int construction* [Joyal, Street & Verity 1996].
2. Make use of the generality of the Int construction and apply it directly to a functional language.
3. Derive a functional language from the resulting structure.

Int Construction

The Int construction can be seen as a general method for turning a model of unidirectional data flow into one with bidirectional data flow.

- GoI situation [Abramsky, Haghverdi, Scott]
- Related to game semantics, context semantics, read-back from optimal reduction, …

Given a 'computation model' $\mathbb{B}$, the Int construction yiels a model Int$(\mathbb{B})$ with bidirectional data flow that is built out of $\mathbb{B}$.

Int Construction

Traced Monoidal category $\mathbb{B}$

- Category $\mathbb{B}$ (here: sets and partial functions)
- Monoidal structure $(+, 0)$ (here: disjoint union)
- Trace (here: while loop)

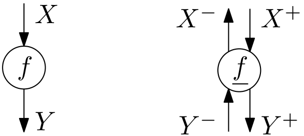
$$\frac{f\colon A + B \longrightarrow C + B}{Tr(f)\colon A \longrightarrow C}$$

$$Tr(f)(a) = Loop(inl(a))$$

$$Loop(x) = \begin{cases} c & \text{if } f(x) = inl(c) \\ Loop(inr(b)) & \text{if } f(x) = inr(b) \end{cases}$$

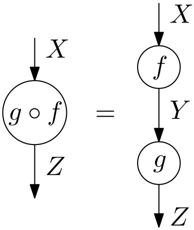
Category Int($\mathbb{B}$)

- Objects are pairs of $\mathbb{B}$-objects $X = (X^-, X^+)$
- Morphism $f\colon X \to Y$ is a $\mathbb{B}$-map $\underline{f}\colon X^+ + Y^- \to Y^+ + X^-$.



Example: Offline Turing Machines appear as morphisms $(\mathit{State} \times \mathbb{N}, \mathit{State} \times \Sigma) \to (\mathbb{N}, \Sigma)$.

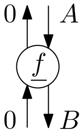
- Composition



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

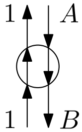
Int($\mathbb{B}$) has well-known structure that allows us to construct 'message passing networks' easily.

- A map from $\underline{f}\colon A \longrightarrow B$ in $\mathbb{B}$ induces a map $(0, A) \longrightarrow (0, B)$ in Int($\mathbb{B}$).



  This gives a full and faithful embedding.

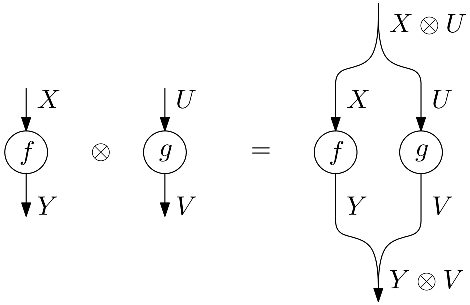
- We shall also use $[A] = (1, A)$, where 1 is a singleton. The value in A is computed only after an explicit request.



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

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

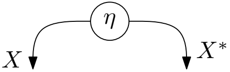
$$(X \otimes Y)^- = X^- + Y^- \qquad I = (0, 0)$$
$$(X \otimes Y)^+ = X^+ + Y^+$$

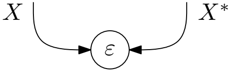


Structure in $\mathrm{Int}(\mathbb{B})$

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

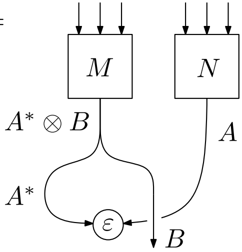
$$(X^-, X^+)^* = (X^+, X^-)$$



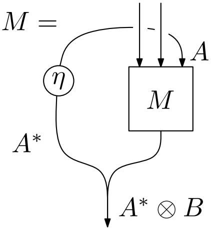


$\mathrm{Int}(\mathbb{B})$ is monoidal closed with $X \multimap Y = X^* \otimes Y$

$M\ N =$



$\lambda x.\, M =$



Structure in $\mathrm{Int}(\mathbb{B})$

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

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

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

Example

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

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

Useful morphism

$$f\colon I \longrightarrow \bigotimes_A [A]$$

given by $\underline{f}\colon A \times 1 \to A \times A$ with $f(a, \langle\rangle) = \langle a, a\rangle$.

Int Construction and Space Complexity

The functions that represent Offline Turing Machines

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

appear in Int(**Pfn**) as morphisms of type

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

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

Int Construction and Space Complexity

The functions that represent Offline Turing Machines

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

appear in Int(**Pfn**) as morphisms of type

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

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

The structure of Int(**Pfn**) is useful for working with OTMs.

- Composition:

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

- 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 Int$(\mathbb{B})$. It can be seen as a *definitional extension* of the initial language.
    4. Identify programs with sublinear space usage.

3. 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

$$min_A \mid succ_A(f) \mid eq_A(f, f)$$

Explicit trace (with respect to $+$)

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

(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

$$\mathtt{x}\colon \alpha,\ \mathtt{y}\colon \alpha \vdash \mathtt{add}(\mathtt{x},\mathtt{y})\colon \alpha$$

With syntactic sugar for tail recursion:

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

With explicit trace:

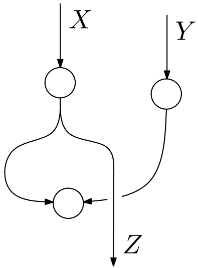
```
add(x, y) =
  (trace p. case p of
      inl(z) -> inr(z)
    | inr(z) -> let z be <x, y> in
       if y = min then inl(x) else inr(<succ x, pred y>)
  ) <x,y>
```

Applying the Int Construction

We apply the Int construction to a *term model* $\mathbb{B}$ of this simple functional language.

We can use the structure in $\mathrm{Int}(\mathbb{B})$ to construct and manipulate message passing networks

- whose nodes are given by terms of the simple language; and
- which are themselves implemented by terms in this language.



corresponds to term of type
$X^+ + Y^+ + Z^- \to Z^+ + X^- + Y^-$

$\Rightarrow$ Definitional extension of the original language.

The Functional Language IntML

IntML extends the simple first order language with syntax for 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 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 Int$(\mathbb{B})$, which are implemented by working class terms.

IntML Type System — Working Class

Usual typing rules, e.g.

$$\frac{\Sigma \vdash f\colon A \qquad \Sigma \vdash g\colon B}{\Sigma \vdash \langle f, g\rangle\colon A \times B}$$

$$\dots$$

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

$$\frac{\Sigma \mid \;\vdash t\colon [A]}{\Sigma \vdash \mathrm{unbox}\, t\colon A}$$

IntML Type System — Upper Class

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

Types

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

(A is a working class type)
In the syntax we write $A \cdot X$ for $\bigotimes_A X$.

Typing Sequents

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

(Σ is a working class context)
The restrictions on the appearance of $\bigotimes_A$ are motivated by Dual Light Affine Logic [Baillot & Terui 2003].

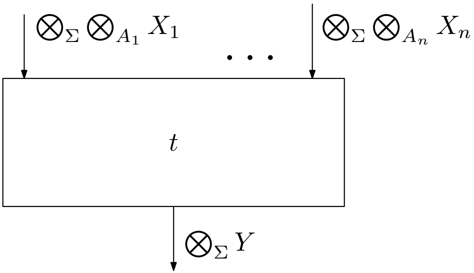
IntML Type System — Upper Class

A sequent

$$\Sigma \mid x_1 \colon A_1 \cdot X_1,\ \ldots, x_n \colon A_n \cdot X_n \vdash t \colon Y$$

denotes morphism

$$\bigotimes_\Sigma \left(\bigotimes_{A_1} X_1 \otimes \cdots \otimes \bigotimes_{A_n} X_n\right) \longrightarrow \bigotimes_\Sigma Y \qquad \text{in } \mathrm{Int}(\mathbb{B}).$$



Upper Class Typing Rules

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

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

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

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

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

(straightforward rules for $\otimes$)

Upper Class Typing Rules

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

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

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

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

Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: [\alpha] \multimap \alpha \cdot [\alpha] \multimap [\alpha]$$

$$g = \lambda f.\, \lambda x.\, \text{let } x \text{ be } [c] \text{ in } f\ [c]\ [c]$$
$$: \alpha \cdot ([\alpha] \multimap [\alpha] \multimap [\beta]) \multimap [\alpha] \multimap [\beta]$$

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

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).

Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \mathsf{let}\ x\ \mathsf{be}\ [c]\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ [d]\ \mathsf{in}\ [\mathtt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha{\times}1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1{\times}\alpha + (\alpha{\times}1 \times \alpha + 1)) \longrightarrow \Sigma \times (1{\times}1 + (\alpha{\times}1 \times 1 + \alpha))$$

Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\ \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1{\times}\alpha + (\alpha{\times}1 \times \alpha + 1)) \longrightarrow \Sigma \times (1{\times}1 + (\alpha{\times}1 \times 1 + \alpha))$$

```
fun x167 -> let (trace fun x166 -> case x166 of inl(x0) -> let x0 be <x5, x0> in case x0 of inl(x4)
-> inr(inr(inr(inl(<x5, x4>)))) | inr(x4) -> inr(inr(inr(inr(inr(inr(inr(inl(<x5, x4>)))))))) |
inr(x165) -> case x165 of inl(x6) -> let x6 be <x11, x10> in inr(inr(inl(<x11, inl(x10)>))) |
inr(x164) -> case x164 of inl(x12) -> inl(x12) | inr(x163) -> case x163 of inl(x18) -> let x18 be
<x23, x18> in let x18 be <x21, x22> in inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inl(<x23, x22>))))
))))))) | inr(x162) -> case x162 of inl(x24) -> let x24 be <x29, x24> in let x24 be <x24, x28> in
let x24 be <x26, x27> in inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inl(<<x29, x26>, <x27,
x28>>))))))))))))) | inr(x161) -> case x161 of inl(x30) -> let x30 be <x35, x34> in inr(inr(inr
(inr(inr(inr(inl(<x35, inl(x34)>))))))) | inr(x160) -> case x160 of inl(x36) -> let x36 be <x41,
x40> in inr(inr(inl(<x41, inr(x40)>))) | inr(x159) -> case x159 of inl(x42) -> let x42 be <x47,
x42> in case x42 of inl(x46) -> inr(inr(inr(inr(inl(<x47, x46>))))) | inr(x46) -> inr(inr(inr(
inr(inr(inr(inr(inr(inr(inl(<x47, x46>)))))))))) | inr(x158) -> case x158 of inl(x48) -> let x48
be <x53, x52> in inr(inr(inr(inr(inr(inr(inl(<x53, inr(x52)>))))))) | inr(x157) -> case x157 of
inl(x54) -> let x54 be <x59, x58> in inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inl(<x59, <>>
)))))))))))) | inr(x156) -> case x156 of inl(x60) -> let x60 be <x65, x64> in inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(<x65, <x64, <>>>)))
)))))))))))))))))))) | inr(x155) -> case x155 of inl(x66) -> let x66 be <x71, x70> in inr(inl(
<x71, <min(* 'a12 *), x70>>)) | inr(x154) -> case x154 of inl(x72) -> let x72 be <x77, x72> in
let x72 be <x75, x76> in inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(...
```

(second half of the term omitted)

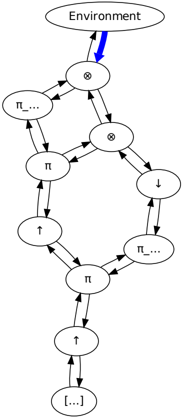
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

`<<>, inr(inr(<>))>`



$\Sigma = 1$
$\alpha = 1 + 1$

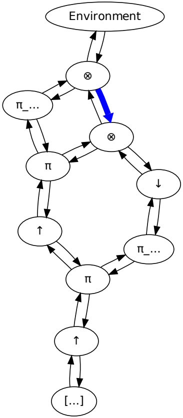
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

`<<>, inr(<>)>`



$\Sigma = 1$
$\alpha = 1 + 1$

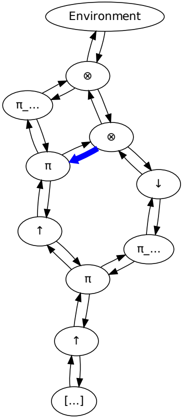
Upper Class Typing — Examples

$$f = \lambda x.\ \lambda y.\ \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

<<>, <>>



$\Sigma = 1$
$\alpha = 1 + 1$

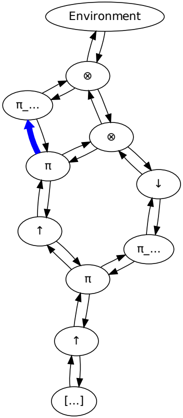
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

<<>, <>>



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

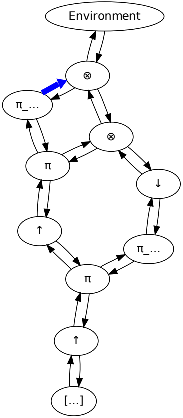
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha{\times}1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1{\times}\alpha + (\alpha{\times}1 \times \alpha + 1)) \longrightarrow \Sigma \times (1{\times}1 + (\alpha{\times}1 \times 1 + \alpha))$$

<<>, <<>, <>>>



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

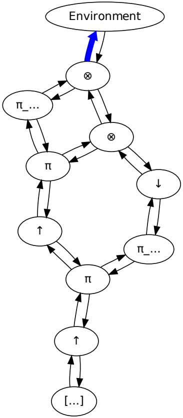
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\ \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

`<<>, inl(<<>, <>>)>`



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

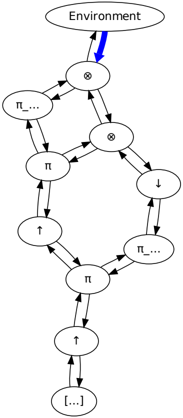
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<>, inl(<<>, inr(<>)>)>
```



$\Sigma = 1$
$\alpha = 1 + 1$

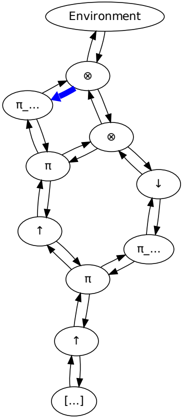
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\ \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<>, <<>, inr(<>)>>
```



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

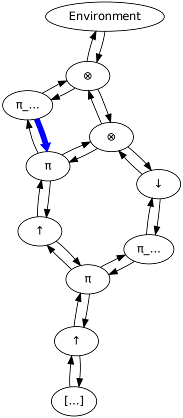
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\ \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\mathtt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<>, inr(<>)>
```



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

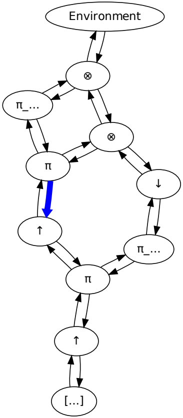
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

<<>, <inr(<>), <>>>



$\Sigma = 1$
$\alpha = 1 + 1$

Upper Class Typing — Examples

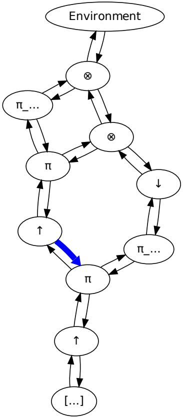
$$f = \lambda x.\ \lambda y.\ \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\mathtt{add}\ c\ d]$$

$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<<>, inr(<>)>, <>>
```



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

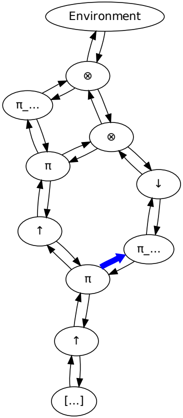
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<<>, inr(<>)>, <>>
```



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

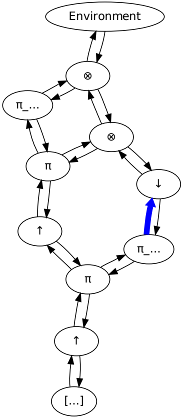
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

`<<<>, inr(<>)>, <<>, <>>>`



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

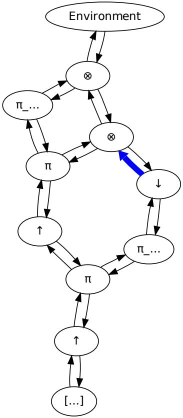
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\; c\; d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<>, <<inr(<>), <>>, <>>>
```



$\Sigma = 1$
$\alpha = 1 + 1$

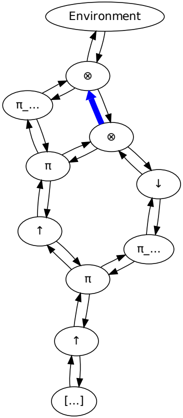
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\ \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<>, inl(<<inr(<>), <>>, <>>)>
```



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

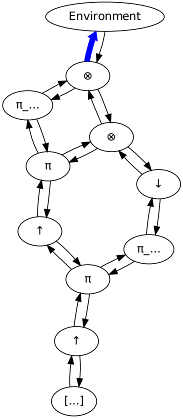
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<>, inr(inl(<<inr(<>), <>>, <>>))>
```



$\Sigma = 1$
$\alpha = 1 + 1$

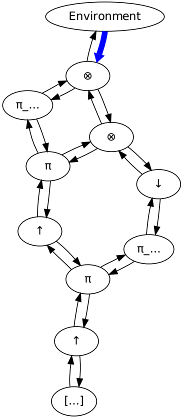
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\ \mathsf{let}\ x\ \mathsf{be}\ [c]\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ [d]\ \mathsf{in}\ [\mathtt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<>, inr(inl(<<inr(<>), <>>, inl(<>)>))>
```



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

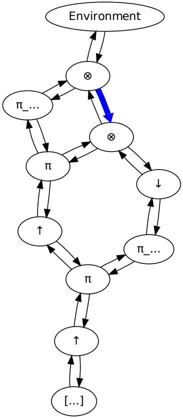
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\mathtt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<>, inl(<<inr(<>), <>>, inl(<>)>)>
```



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

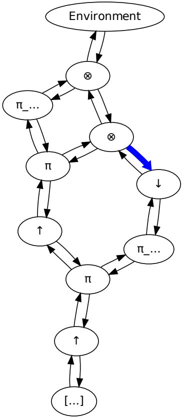
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<>, <<inr(<>), <>>, inl(<>)>>
```



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

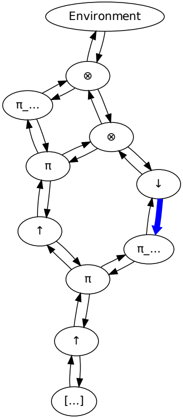
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

`<<<>, inr(<>)>, <<>, inl(<>)>>`



$\Sigma = 1$
$\alpha = 1 + 1$

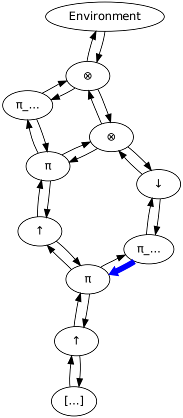
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

`<<<>, inr(<>)>, inl(<>)>`



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

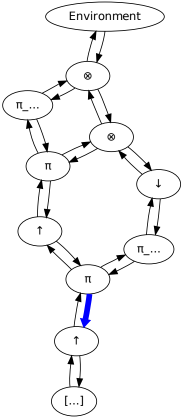
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<<>, inr(<>)>, <inl(<>), <>>>
```



$\Sigma = 1$
$\alpha = 1 + 1$

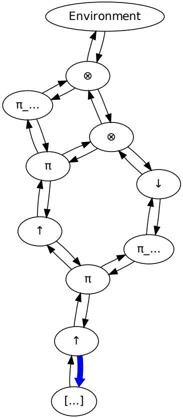
Upper Class Typing — Examples

$$f = \lambda x.\ \lambda y.\ \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<<<>, inr(<>)>, inl(<>)>, <>>
```



$\Sigma = 1$
$\alpha = 1 + 1$

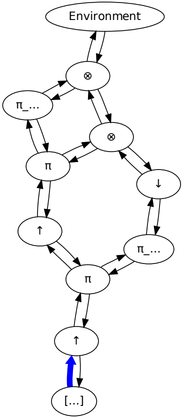
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<<<>, inr(<>)>, inl(<>)>, inr(<>)>
```



$\Sigma = 1$
$\alpha = 1 + 1$

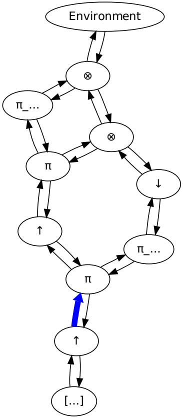
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<<>, inr(<>)>, <inl(<>), inr(<>)>>
```



$\Sigma = 1$
$\alpha = 1 + 1$

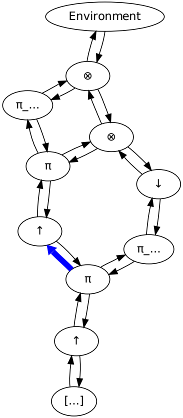
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<<>, inr(<>)>, inr(<>)>
```



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

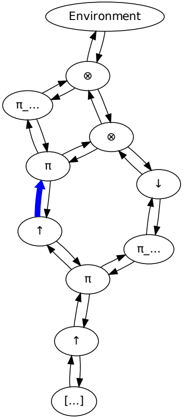
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1{\times}\alpha + (\alpha{\times}1 \times \alpha + 1)) \longrightarrow \Sigma \times (1{\times}1 + (\alpha{\times}1 \times 1 + \alpha))$$

```
<<>, <inr(<>), inr(<>)>>
```



$\Sigma = 1$
$\alpha = 1 + 1$

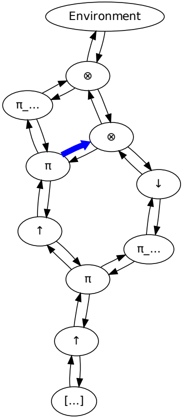
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\; c\; d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

`<<>, inr(<>)>`



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

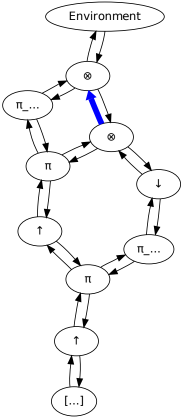
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\ \mathrm{let}\ x\ \mathrm{be}\ [c]\ \mathrm{in}\ \mathrm{let}\ y\ \mathrm{be}\ [d]\ \mathrm{in}\ [\mathtt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha \times 1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1 \times \alpha + (\alpha \times 1 \times \alpha + 1)) \longrightarrow \Sigma \times (1 \times 1 + (\alpha \times 1 \times 1 + \alpha))$$

```
<<>, inr(inr(<>))>
```



$\Sigma = 1$
$\alpha = 1 + 1$

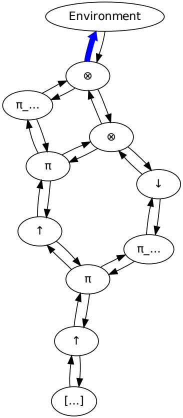
Upper Class Typing — Examples

$$f = \lambda x.\, \lambda y.\, \text{let } x \text{ be } [c] \text{ in let } y \text{ be } [d] \text{ in } [\texttt{add}\ c\ d]$$
$$: 1 \cdot [\alpha] \multimap (\alpha{\times}1) \cdot [\alpha] \multimap [\alpha]$$

represents a working-class term of type

$$\Sigma \times (1{\times}\alpha + (\alpha{\times}1 \times \alpha + 1)) \longrightarrow \Sigma \times (1{\times}1 + (\alpha{\times}1 \times 1 + \alpha))$$

```
<<>, inr(inr(inr(<>)))>
```



$$\Sigma = 1$$
$$\alpha = 1 + 1$$

Hacking

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

Hacking

Have we captured all the structure of Int$(\mathbb{B})$?
No! Int$(\mathbb{B})$ has a lot more structure!

Can we ever capture all the useful structure?

Hacking

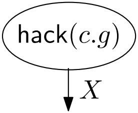
Have we captured all the structure of Int$(\mathbb{B})$?

No! 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!

$$(\text{Hack})\ \frac{\Sigma,\ c\colon X^- \vdash g\colon X^+}{\Sigma \mid \Gamma \vdash \mathsf{hack}(c.g)\colon X}$$



$$\begin{aligned}[A]^- &= 1 \\ (X \otimes Y)^- &= X^- + Y^- \\ (A \cdot X \multimap Y)^- &= A \times X^+ + Y^-\end{aligned} \qquad \begin{aligned}[A]^+ &= A \\ (X \otimes Y)^+ &= X^+ + Y^+ \\ (A \cdot X \multimap Y)^+ &= A \times X^- + Y^+\end{aligned}$$

Complexity results remain true in presence of (Hack).

Hacking — `loop`

$$\texttt{loop}\colon \alpha \cdot (\gamma \cdot [\alpha] \multimap [\alpha + \beta]) \multimap [\alpha] \multimap [\beta]$$

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

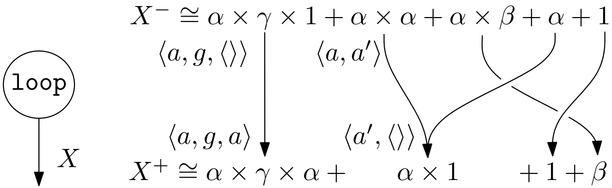
```
loop = hack x. case x of
      | inl(y) -> let y be <store, stepq> in
            case stepq of
            | inl(argq) -> let argq be <argstore, unit> in
                               inl(<store, inl(<argstore, store>)>)
            | inr(contOrStop) -> case contOrStop of
                                 | 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 — `loop`

$$\texttt{loop}\colon \alpha \cdot (\gamma \cdot [\alpha] \multimap [\alpha + \beta]) \multimap [\alpha] \multimap [\beta]$$

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

$$X = \alpha \cdot (\ \gamma\ \cdot\ [\alpha] \multimap [\alpha + \beta]) \multimap [\alpha] \multimap [\beta]$$



Functional Programming in Sublinear Space

1. Computation with external data

2. Deriving the functional language IntML

3. Programming in IntML

   How easy is it to write sublinear space algorithms in IntML?

   Consider two known LOGSPACE algorithms:

   - Checking acyclicity in undirected graphs
   - Recursion by computational amnesia

Functional Programming in Sublinear Space

1. Computation with external data

2. Deriving the functional language IntML

3. Programming in IntML

   How easy is it to write sublinear space algorithms in IntML?

   Consider two known LOGSPACE algorithms:

   - Checking acyclicity in undirected graphs
   - Recursion by computational amnesia

Graph Algorithms

Represent graphs by upper class types of the form

$$G(\alpha) = ([\alpha] \to [2]) \otimes ([\alpha \times \alpha] \to [2]) \,.$$

- Carrier set: α
- Set of nodes: $[\alpha] \to [2]$
- Edge relation: $[\alpha \times \alpha] \to [2]$

We often omit the index type, writing just $X \to Y$ for $A \cdot X \multimap Y$.

Graph Algorithms

Represent graphs by upper class types of the form

$$G_{\beta,\gamma}(\alpha) = (\beta \cdot [\alpha] \multimap [2]) \otimes (\gamma \cdot [\alpha \times \alpha] \multimap [2])\,.$$

- Carrier set: α
- Set of nodes: $\beta \cdot [\alpha] \multimap [2]$
- Edge relation: $\gamma \cdot [\alpha \times \alpha] \multimap [2]$

We often omit the index type, writing just $X \to Y$ for $A \cdot X \multimap Y$.

Graph Algorithms

Represent graphs by upper class types of the form

$$G(\alpha) = ([\alpha] \to [2]) \otimes ([\alpha \times \alpha] \to [2])\,.$$

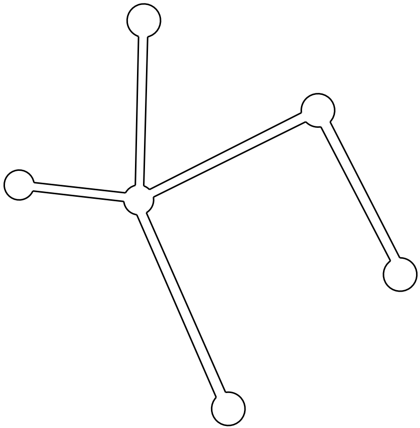
Suppose we write an upper class term containing only the type variable α. Its translation to a working class term then has a working-class type containing only the variable α.

- For any working class type $P(\alpha)$ there are constants k and l such that, for any closed type A with n values, the type $P(A)$ has at most $n^k + l$ values.
- If A is a closed type with n values then $G(A)$ can stand for graphs of size n.

$\Rightarrow$ With a binary encoding, tokens can be stored in logarithmic space.

LOGSPACE Algorithm for Checking Acyclicity of Undirected Graphs

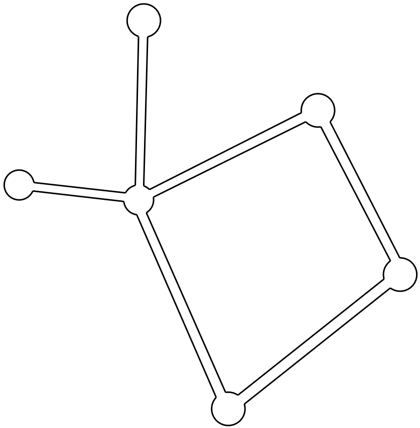
Check that *any* walk according to the right-hand-rule returns to the node it started from through the edge over which it left.
[Cook & McKenzie 1980]



LOGSPACE Algorithm for Checking Acyclicity of Undirected Graphs

Check that *any* walk according to the right-hand-rule returns to the node it started from through the edge over which it left.

[Cook & McKenzie 1980]



Walk following the Right-Hand-Rule

$$
\begin{array}{l}
\mathit{checkpath}_{M,\alpha} \colon G(\alpha) \to [\alpha] \to [\alpha] \to [2] \\
\mathit{checkpath}_{M,\alpha} = \lambda \mathit{graph}.\, \lambda \mathit{ie}. \\
\quad \text{copy } \mathit{graph} \text{ as } \mathit{graph}_1, \mathit{graph}_2 \text{ in} \\
\quad \text{let } \mathit{ie} \text{ be } [e] \text{ in} \\
\quad \mathit{if2}\ (\mathit{edge}\ \mathit{graph}_1\ [e]) \\
\qquad (\texttt{loop}\ (\lambda w.\ \text{let } w \text{ be } [p] \text{ in} \\
\qquad\qquad\qquad \mathit{if2}\ [\mathit{dst}\ p = \mathit{src}\ e] \\
\qquad\qquad\qquad\quad (\mathit{if2}\ [\mathit{src}\ p = \mathit{dst}\ e]\ [\text{return}(\mathit{true})]\ [\text{return}(\mathit{false})]) \\
\qquad\qquad\qquad\quad (\text{let } \mathit{nextEdge}\ \mathit{graph2}\ [p] \text{ be } [d] \text{ in} \\
\qquad\qquad\qquad\qquad [\text{continue}(\langle \mathit{dst}\ p, d \rangle)]) \\
\qquad\qquad)\ [e] \\
\qquad [\mathit{true}]
\end{array}
$$

Using abbreviations like $\mathit{src}\ e = \pi_1\ e$ and $\mathit{return}\ x = \mathit{inr}(x)$ and defined functions like *if2*.

Checking Acyclicity

$$
\begin{aligned}
\textit{iterator}_\alpha &: ([\alpha] \to [\beta]) \to ([\beta] \to [\beta] \to [\beta]) \to [\beta] \\
\textit{iterator}_\alpha &= \lambda x.\, \lambda y.\, \text{copy } x \text{ as } x_1, x_2 \text{ in} \\
&\qquad \texttt{loop}\ (\lambda w.\, \text{let } w \text{ be } [e] \text{ in} \\
&\qquad\qquad \textit{if2}\ [\pi_2\, e = \textit{max}]\ [\text{return}(\pi_1\, e)] \\
&\qquad\qquad\quad (\text{let } y\ (x_1\ [\textit{succ}(\pi_2\, e)])\ [\pi_1\, e] \text{ be } [f] \text{ in} \\
&\qquad\qquad\qquad [\text{continue}\langle f, (\textit{succ}(\pi_2\, e))\rangle])) \\
&\qquad (\text{let } x_2\ [\textit{min}] \text{ be } [f] \text{ in } [\langle f, \textit{min}\rangle])
\end{aligned}
$$

$$
\begin{aligned}
\textit{checkcycle}_{M,\alpha} &: G(\alpha) \to [2] \\
\textit{checkcycle}_{M,\alpha} &= \lambda \textit{graph}. \\
&\qquad \text{copy } \textit{graph} \text{ as } \textit{graph}_1, \textit{graph}_2 \text{ in} \\
&\qquad \textit{iterator}_{\alpha\times\alpha}\ (\lambda \textit{ie}.\, \text{let } \textit{ie} \text{ be } [e] \text{ in} \\
&\qquad\qquad \textit{if2}\ (\textit{edge}\ \textit{graph}_1\ [e]) \\
&\qquad\qquad\quad (\textit{checkpath}_{M,\alpha}\ \textit{graph}_2\ [e]) \\
&\qquad\qquad\quad [\textit{true}]) \\
&\qquad\quad \textit{and}
\end{aligned}
$$

Checking Acyclicity

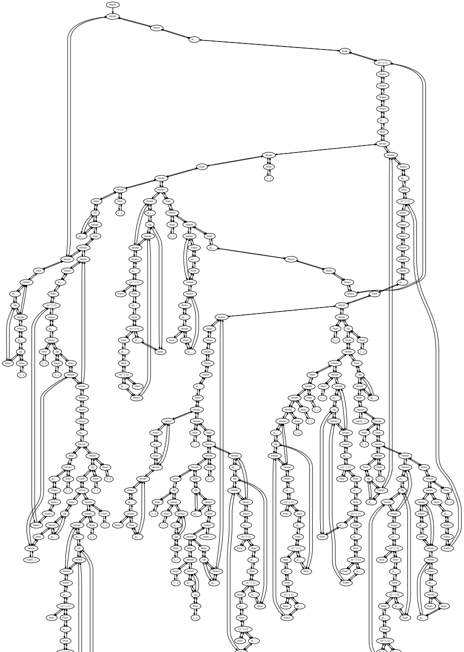
The type of $\mathit{checkcycle}_{M,\alpha}$ may be expanded to

$$A \cdot (([\alpha] \multimap [2]) \otimes ([\alpha \times \alpha] \multimap [2])) \multimap [2],$$

where A is the following type:

$$((2\times(\alpha\times\alpha)\times(2\times(\alpha\times\alpha)\times(2\times\alpha_1\times(\alpha_2\times\alpha_3)))+1\times\alpha_4)\times\alpha_5\times(\alpha\times\alpha\times(\alpha_6\times(\alpha_7\times\alpha_8\times\alpha_9\times\alpha_{10})))+(2\times(\alpha\times\alpha)\times(2\times(\alpha\times\alpha)\times(2\times\alpha_1\times(\alpha_2\times\alpha_3)))+1\times\alpha_4)\times\alpha_5\times(\alpha\times\alpha\times(2\times\alpha_{11}\times((\alpha\times\alpha\times(\alpha_{12}\times(\alpha_{13}\times\alpha_{14}\times\alpha_{15}\times\alpha_{16}))+\alpha\times\alpha\times(2\times\alpha_{17}\times(\alpha\times\alpha\times(\alpha\times\alpha\times(2\times\alpha_{21}\times(\alpha\times(\alpha\times(\alpha_{22}\times(\alpha_{23}\times\alpha_{24}\times\alpha_{25}\times\alpha_{26})))))))))\times\alpha_{18}\times\alpha_{19}))))\times\alpha_{20}$$

Checking Acyclicity



Checking Acyclicity — Space Usage

An upper bound for the space needed to evalueate $checkcycle_{M,\alpha}$ can be read off from the network it compiles to.

To answer requests, we need to store

- the network that $checkcycle_{M,\alpha}$ compiles to;
- one token on this graph and its position.

Each edge of the network is annotated with a type $A(\alpha)$, so the size of a token can be bounded statically by looking at all the types that appear in the network.

- Given a graph of size n, we choose for α the type $2 \times \cdots \times 2$ ($\log n + 1$ times), which is large enough to represent the graph.
- The size of values of $A(\alpha)$ is at most $k \cdot \log n + l$.

$\Rightarrow$ Token uses logarithmic space.

$\Rightarrow$ Since network size is a constant, IntML-evaluation of $checkcycle_{M,\alpha}$ therefore remains in LOGSPACE.

A Functional Language for Sublinear Space

1. Computation with external data

2. Deriving the functional language IntML

3. Programming in IntML

   How easy is it to write sublinear space algorithms in IntML?

   Consider two known LOGSPACE algorithms:

   - Checking acyclicity in undirected graphs
   - Recursion by computational amnesia

Working with Binary Strings

Represent binary strings by the upper class type

$$\mathsf{S}(\alpha) = [\alpha] \to [3],$$

where the type 3 contains elements for 0, 1 and a blank symbol.

Goal:
Implement (higher-order) combinators on strings that allow us to work with large strings as if we had enough memory to store them.

Simple Examples

Empty String

$$\begin{aligned} \texttt{zero} :&\quad [\alpha] \to [3] \\ \texttt{zero} :=&\quad \lambda w.\, [\mathit{blank}] \end{aligned}$$

Appending '0'

$$\begin{aligned} \texttt{succ}_0 :&\quad ([\alpha] \to [3]) \to ([\alpha] \to [3]) \\ \texttt{succ}_0 :=&\quad \lambda w.\, \lambda i.\, \text{let } i \text{ be } [c] \text{ in} \\ &\qquad\qquad \text{case } (c = \mathit{min}) \text{ of inl}(\mathit{true}) \Rightarrow [\mathit{zero}] \\ &\qquad\qquad\qquad\qquad\qquad \mid \text{inr}(\mathit{false}) \Rightarrow w\, [\mathit{pred}\ c] \end{aligned}$$

Case distinction

$$\begin{aligned} \texttt{if} :&\quad ([\alpha] \to [3]) \to ([\alpha] \to [3]) \to ([\alpha] \to [3]) \to ([\alpha] \to [3]) \\ \texttt{if} :=&\quad \lambda w.\, \lambda w_0.\, \lambda w_1.\, \lambda i.\, \text{let } w\, [\mathit{min}] \text{ be } [c] \text{ in} \\ &\qquad \text{case } c \text{ of inl}(\mathit{blank}) \Rightarrow w_0\ i \\ &\qquad\qquad\quad \mid \text{inr}(\mathit{zo}) \Rightarrow \text{case } \mathit{zo} \text{ of inl}(\mathit{zero}) \Rightarrow w_0\ i \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{inr}(\mathit{one}) \Rightarrow w_1\ i \end{aligned}$$

Simple Examples

Such combinators can be used for working with large words, e.g.

$$\lambda w.\, \lambda v.\, \mathtt{succ}_0\, (\mathtt{if}\; w\, (\mathtt{succ}_0\, \mathtt{zero})\, v)$$

So far, the combinators are very simple.

Can interesting combinators can be implemented in this way?

Function Algebras

BC^- [Murawski, Ong] and BC^-_ε [Møller-Neergaard]

Variants of primitive recursion on binary words that can be evaluated in LOGSPACE.

- Example basic functions:

$$succ_0(: y) = y0$$

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

- Closed under composition
- Closed under (course-of-value) recursion on notation:
  $f = saferec(g, h_0, h_1, d_0, d_1)$ 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^- and BC^-_ε

Møller-Neergaard proves LOGSPACE-soundness by implementing BC^-_ε in SML/NJ:

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

$$(\mathbb{N} \to 3) \to \cdots \to (\mathbb{N} \to 3) \to (\mathbb{N} \to 3) \quad .$$

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

Implementing Recursion by Computational Amnesia

$$\begin{aligned} g &: (\mathbb{N} \to 3) \\ h_0 &: (\mathbb{N} \to 3) \to (\mathbb{N} \to 3) \\ h_1 &: (\mathbb{N} \to 3) \to (\mathbb{N} \to 3) \end{aligned}$$

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

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

- 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.

$\Rightarrow$ 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
    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
                  h x' x2 ... xm (recursiveCall d) (#bt (!goal))
              end
          in ( result := { depth = #depth (!goal),
                           res = res,
                           bt = #bt (!goal) };
               true )
          end handle Restartn => false
        0 = #depth (!result) ));
      #res (!result))
    end
```

Control Flow

$$\texttt{callcc}\colon (\gamma \cdot ([\alpha] \multimap \beta) \multimap [\alpha]) \multimap [\alpha]$$

Implemented using hack:

$$(\gamma \times (\alpha + \beta^-) + \alpha) + 1 \longrightarrow (\gamma \times (1 + \beta^+) + 1) + \alpha$$

$$\begin{aligned} \mathit{inr}(*) &\mapsto \mathit{inl}(\mathit{inr}(*)) \\ \mathit{inl}(\mathit{inr}(a)) &\mapsto \mathit{inr}(a) \\ \mathit{inl}(\mathit{inl}(g, \mathit{inr}(x))) &\mapsto \mathit{inl}(\mathit{inl}(g, \mathit{inl}(*))) \\ \mathit{inl}(\mathit{inl}(g, \mathit{inl}(a))) &\mapsto \mathit{inr}(a) \end{aligned}$$

SML

```
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
    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
                  h x' x2 ... xm (recursiveCall d) (#bt (!goal))
              end
          in ( result := { depth = #depth (!goal),
                           res = res,
                           bt = #bt (!goal) };
               true )
          end handle Restartn => false
        0 = #depth (!result) ));
      #res (!result))
    end
```

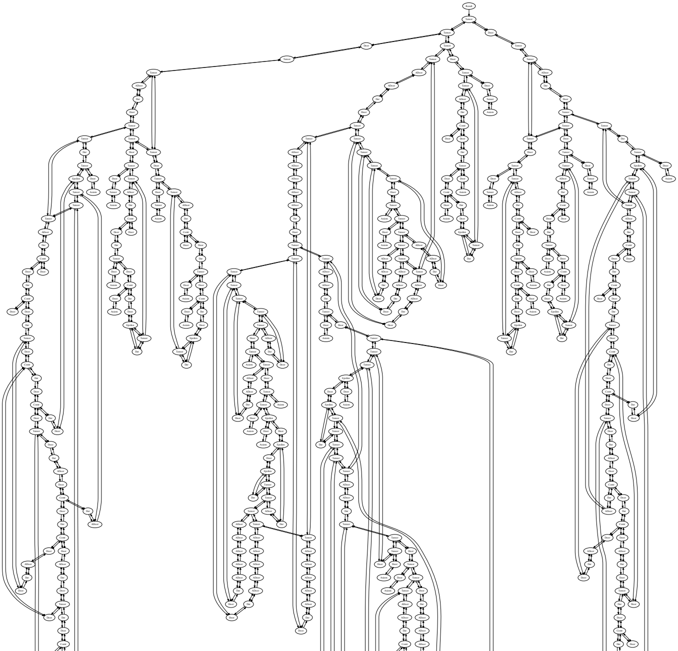
IntML

```
recursiveCall =U
  fun k -> fun state -> fun x ->
    let x be [xc] in
    let state be [statec] in
    case and (succ (goaldepth statec) = (resultdepth statec)) (xc = (resultbit s
    of
      inl(true) -> [resultres statec]
    | inr(false) -> k [inl(<result statec, <xc, succ (goaldepth statec)>>)];

shiftarg =U fun x -> fun shift ->
  fun i ->
    let i be [ic] in
    let shift be [shiftc] in
      x [add ic shiftc];

innerloopbody =U
fun g -> fun h0 -> fun h1 -> fun x ->
fun state -> fun k ->
  copy x as x1, x23 in
  copy x23 as x2, x3 in
  copy k as k1, k2 in
    let state be [statec] in
    let if3 (x1 [goaldepth statec])
              (g [goalbit statec])
              (h0 (shiftarg x2 [goaldepth statec])
                  (recursiveCall k1 [statec]) [goalbit statec])
              (h1 (shiftarg x3 [goaldepth statec])
                  (recursiveCall k2 [statec]) [goalbit statec]) be [res]
      in [inr(<<<goaldepth statec, res>, goalbit statec>, goal statec>)];

innerloop =U
fun g -> fun h0 -> fun h1 -> fun x1 -> fun bit ->
fun res ->
    let res be [resc] in
    let bit be [bitc] in
    let loop (fun state -> callcc (fun k -> innerloopbody g h0 h1 x1 state k))
             [<resc, <bitc, min> (*goal*)>] be [finalstate] in
      [if (resultdepth finalstate) = min then
        inr(result finalstate)
       else inl(result finalstate)];

saferec =U
fun g -> fun h0 -> fun h1 -> fun x1 -> fun bit ->
  let
  loop (innerloop g h0 h1 x1 bit) [<<max, min>, min>]
  be [res] in [resultbit res];
```

Example — saferec

$$
\begin{aligned}
\texttt{parity} &= \lambda w.\, \texttt{saferec}\,\texttt{zero}\, h_0\, h_1\, w \\
h_0 &= \lambda x.\, \lambda v.\, \texttt{if}\; v\, (\texttt{succ}_1\, \texttt{zero})\, (\texttt{succ}_0\, \texttt{zero}) \\
h_1 &= \lambda x.\, \lambda v.\, \texttt{if}\; v\, (\texttt{succ}_0\, \texttt{zero})\, (\texttt{succ}_1\, \texttt{zero}) \\
f &= \lambda w.\, \texttt{parity}\, (\texttt{succ}_0\, \texttt{succ}_1\, w)
\end{aligned}
$$

String Diagram for `parity`



Working Class Term for `parity`

```
let (trace x9472. case x9472 of inl(x0)  -> let x0 be <x5, x4> in inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
...
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(
inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inr(inl(<x5, inr(x4)>)))))))))))))))
))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
...
```

…5 Megabyte more

(about 150 Kb if injections were represented efficiently)

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];
```

LOGSPACE Soundness and Completeness

Any upper class term

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

represents a LOGSPACE function on binary words and any such function can be represented in this way.

If we consider α as a natural number then t induces a function

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

as follows:

- A word $w \in \{0,1\}^{\leq\alpha}$ can be represented as a function in $\langle w\rangle\colon B\cdot[\alpha] \multimap [3]$ 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 and Completeness

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

Step function

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

$$\begin{aligned} \mathit{step} = \lambda x.\quad & \text{let } x \text{ be } [s] \text{ in} \\ & \text{let } \mathit{input}\ [\mathit{inputpos}(s)] \text{ be } [i] \text{ in} \\ & [\ldots\text{working class term for transition function } \ldots] \end{aligned}$$

Turing Machine

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

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

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 should allow us to work with polylogarithmic space.
- Completeness: Can we get completeness by something less trivial than `hack`?