The Complexity of Interaction

Stéphane Gimenez
Georg Moser

University of Innsbruck
Computational Logic Group

DICE — 3rd April 2016

Introduction

- Is it possible to efficiently run in parallel uninstrumented functional programs?

```
                    let msort l f =
let a, b = split l in merge f (msort a) (msort b)
```

Introduction

- Is it possible to efficiently run in parallel uninstrumented functional programs?

```
                    let msort l f =
let a, b = split l in merge f (msort a) (msort b)
```

- How to know the resource needs (time/space) of program parts?

Introduction

- Is it possible to efficiently run in parallel uninstrumented functional programs?

```
                    let msort l f =
let a, b = split l in merge f (msort a) (msort b)
```

- How to know the resource needs (time/space) of program parts?
- User-provided formal complexity specifications:

$$\texttt{msort} :: \text{list}\, A \to (A \to A \to \text{bool}) \to \text{list}\, A$$

Introduction

- Is it possible to efficiently run in parallel uninstrumented functional programs?

```
                    let msort l f =
let a, b = split l in merge f (msort a) (msort b)
```

- How to know the resource needs (time/space) of program parts?
- User-provided formal complexity specifications:

$$\texttt{msort} :: \mathrm{list}_n A \multimap !_{n \log n}(A \multimap A \multimap \mathrm{bool}) \multimap \mathrm{list}_n A$$

Introduction

- Is it possible to efficiently run in parallel uninstrumented functional programs?

```
                    let msort l f =
let a, b = split l in merge f (msort a) (msort b)
```

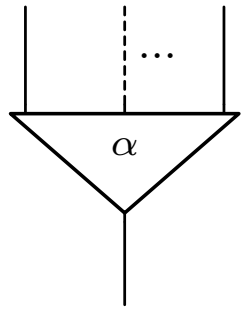
- How to know the resource needs (time/space) of program parts?
- User-provided formal complexity specifications:

$$\texttt{msort} :: \mathrm{list}_n A \multimap !_{n \log n}(A \multimap A \multimap \mathrm{bool}) \multimap \mathrm{list}_n A$$
$$\text{time: } \tau(n), \text{ space: } \sigma(n)$$

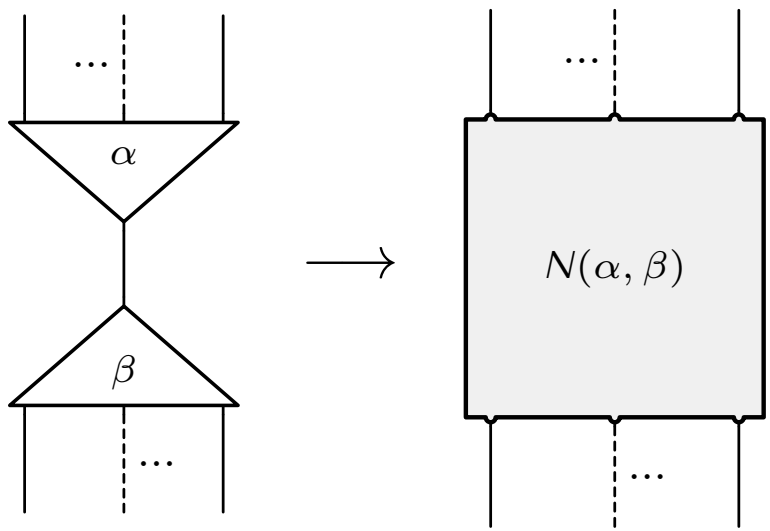
Outline

Interaction Nets

Yves Lafont, POPL 1990. A user-defined set of symbols, with arities.

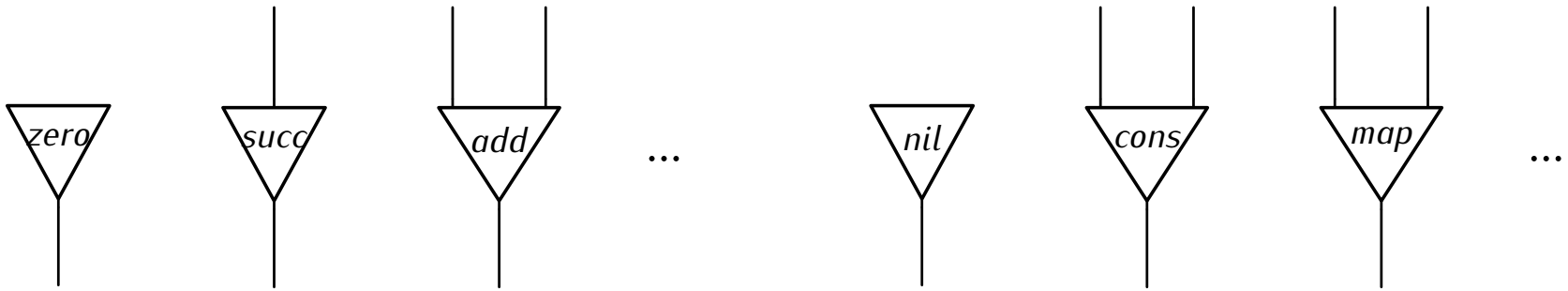


A user-defined set of reduction rules. By design, interaction-net redexes consist of two nodes only:

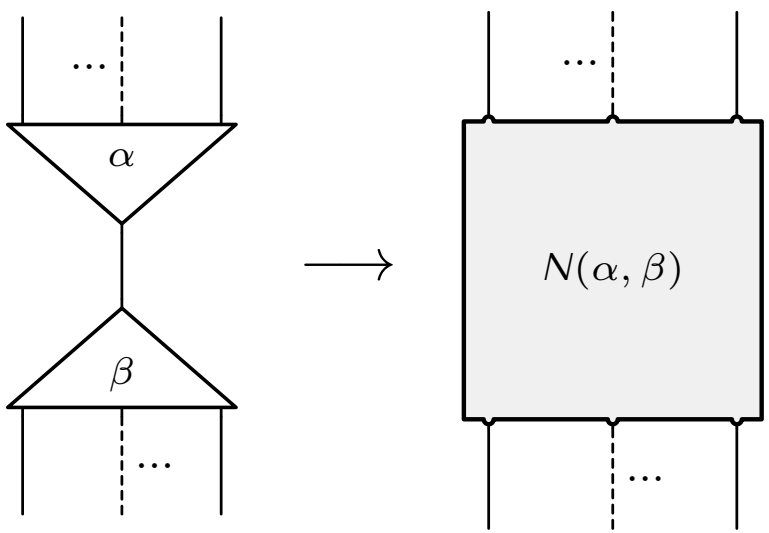


Interaction Nets

Yves Lafont, POPL 1990. A user-defined set of symbols, with arities.

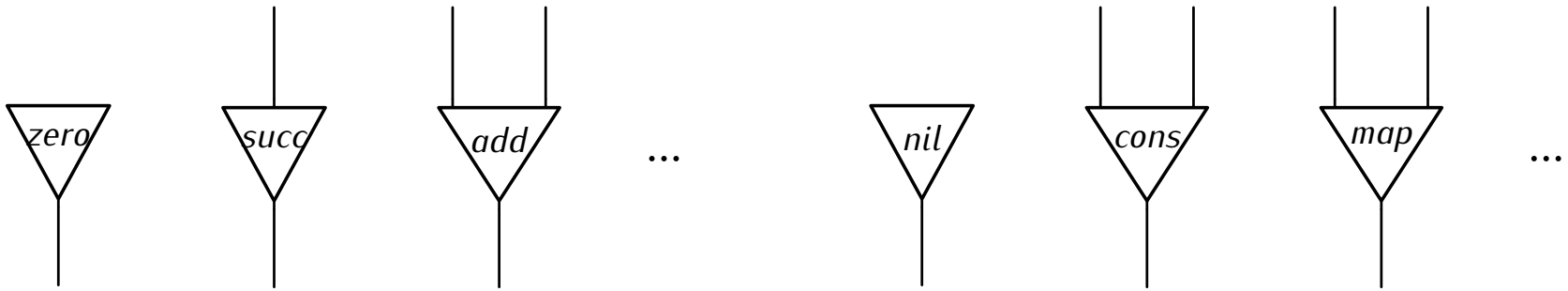


A user-defined set of reduction rules. By design, interaction-net redexes consist of two nodes only:

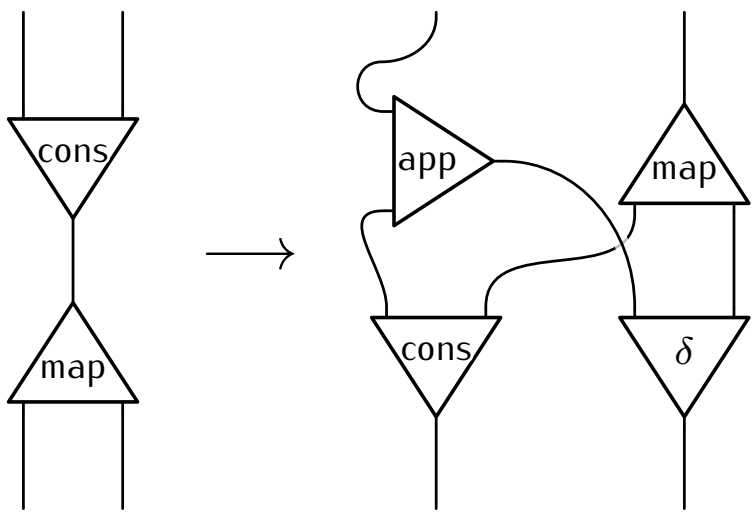


Interaction Nets

Yves Lafont, POPL 1990. A user-defined set of symbols, with arities.

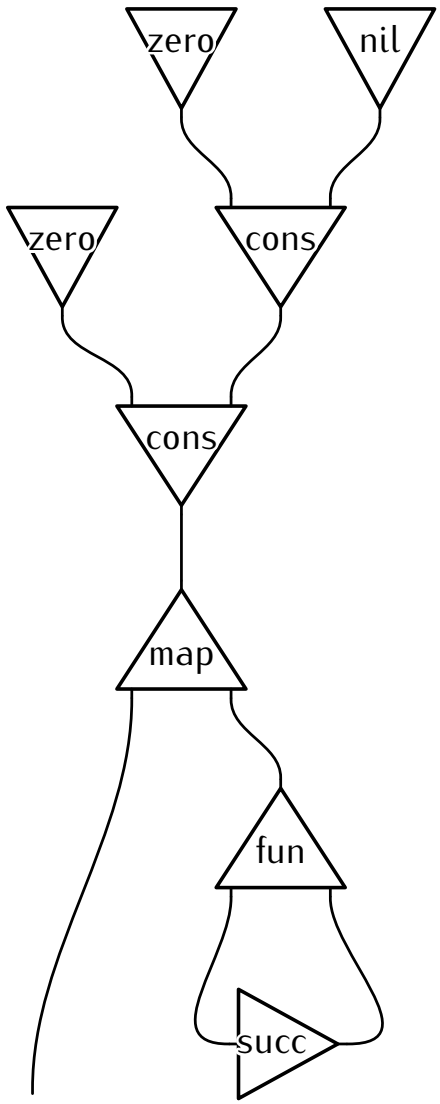


A user-defined set of reduction rules. By design, interaction-net redexes consist of two nodes only:

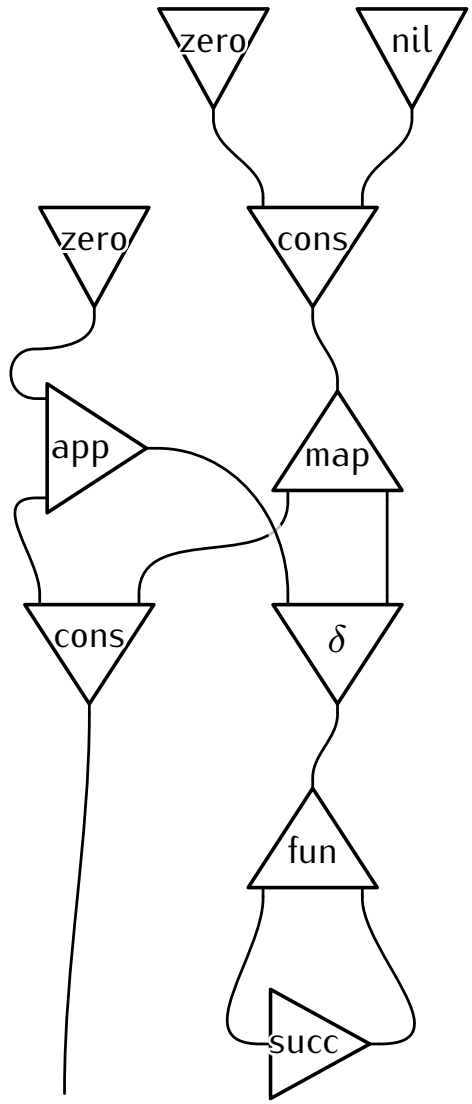


A program with inputs:

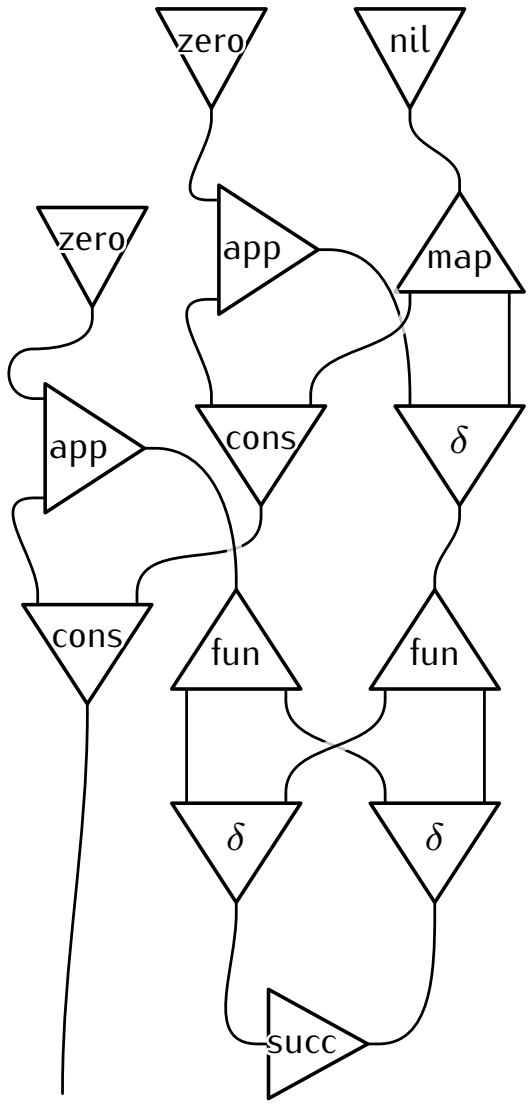
```
List.map (fun x -> Succ x) [Zero; Zero]
```



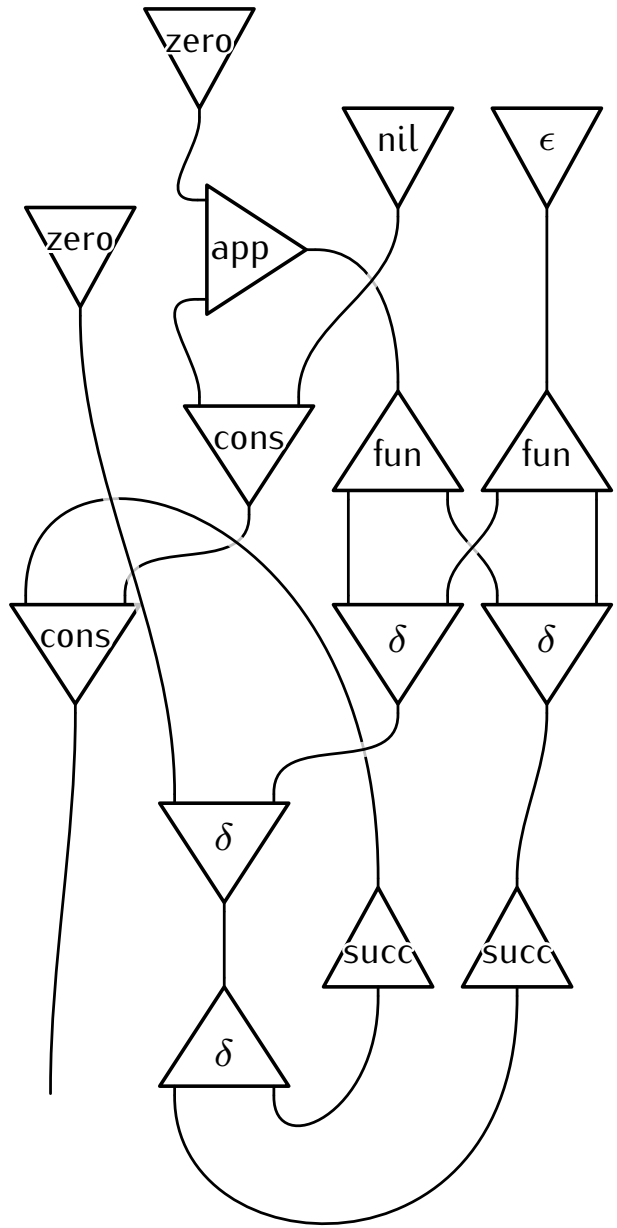
A program with inputs:



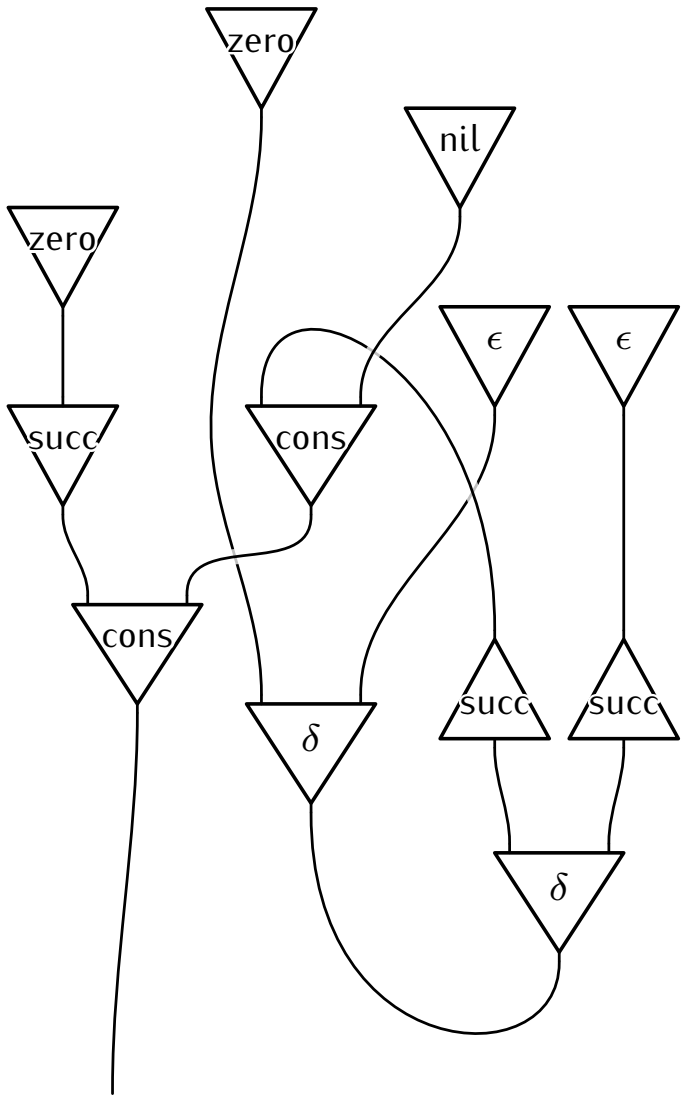
A program with inputs:



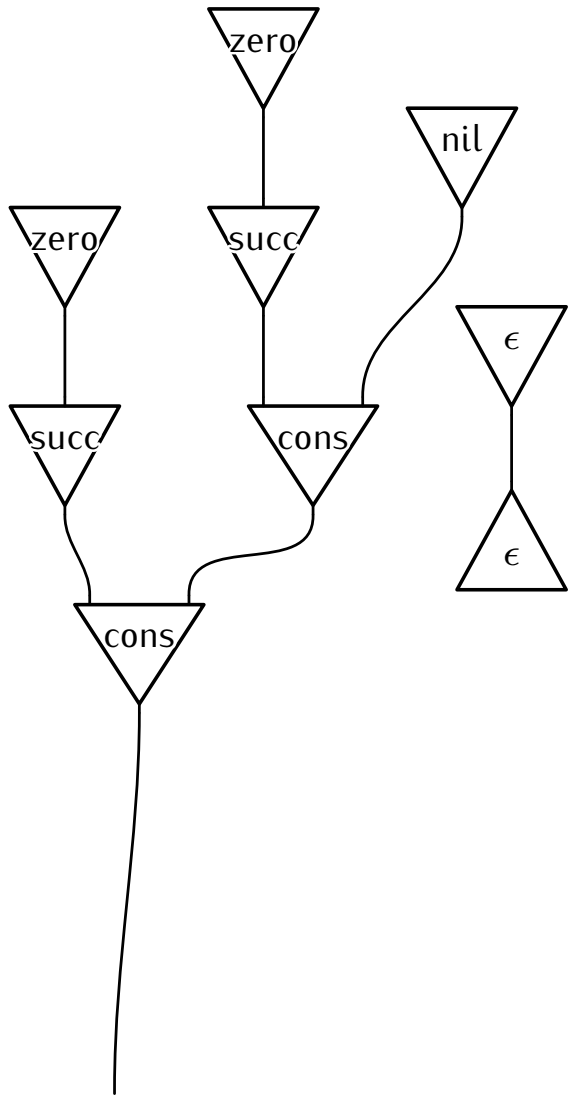
A program with inputs:



A program with inputs:

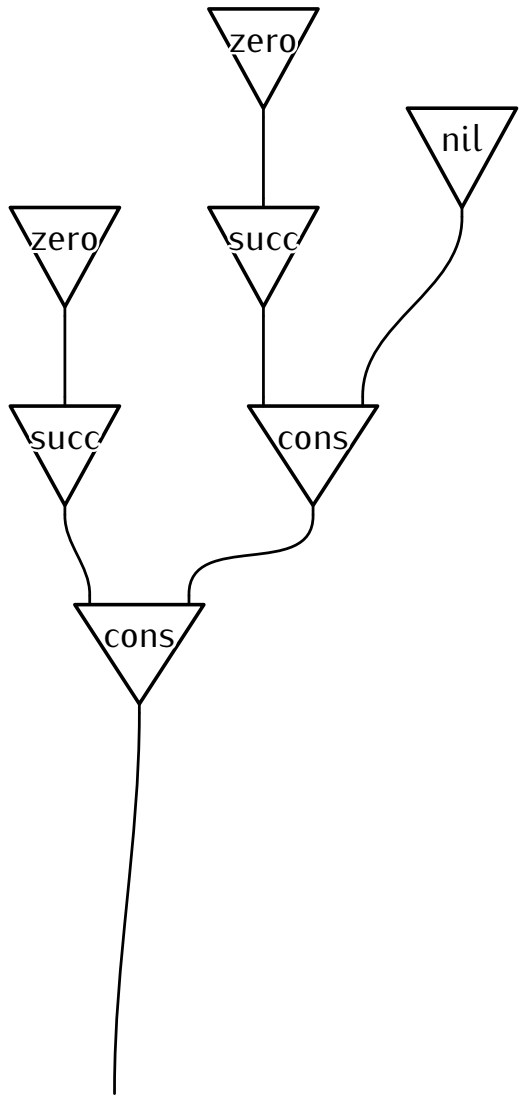


A program with inputs:



A program with inputs:

```
[Succ Zero; Succ Zero]
```



Properties:

- *Sequential Reduction* (CPU): Satisfies the diamond property.
- *Parallel Reduction* (GPU, Customized electronic chips, Network): Strictly deterministic in the ideal case.
- *Realistic model*: The locating, recording and firing of a redex can all be implemented with bounded resources on a standard computer.

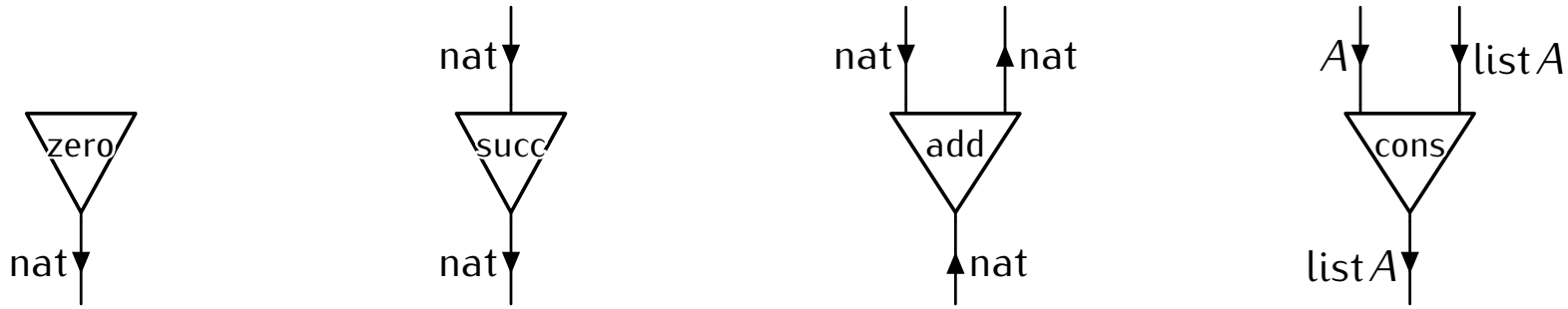
Properties:

- *Sequential Reduction* (CPU): Satisfies the diamond property.
- *Parallel Reduction* (GPU, Customized electronic chips, Network): Strictly deterministic in the ideal case.
- *Realistic model*: The locating, recording and firing of a redex can all be implemented with bounded resources on a standard computer.

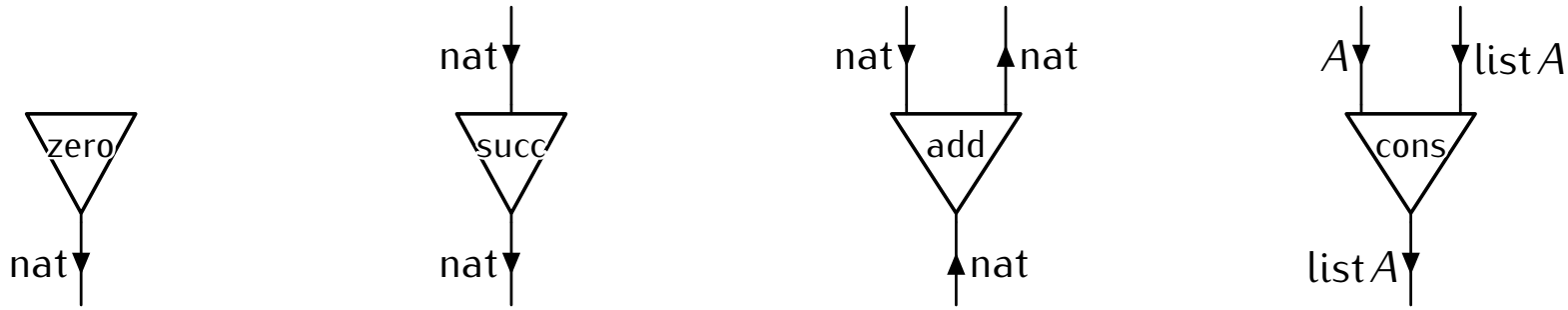
Resource costs:

- Time:
  Number of reduction steps / Sum of step durations: $L \xrightarrow{d} R$.
- Space:
  Number of nodes / Sum of node weights: $|c|$.

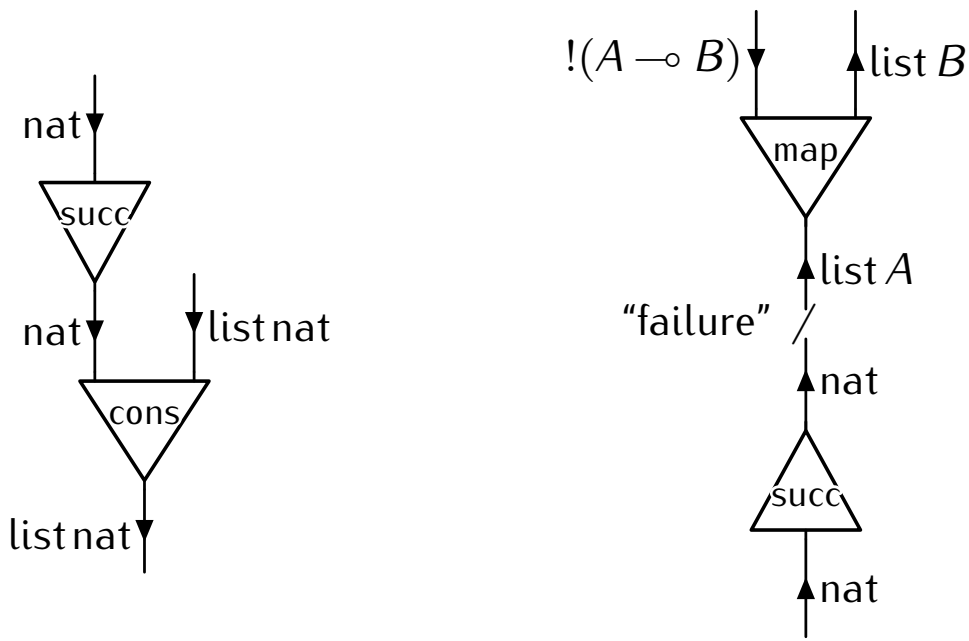
Typing:



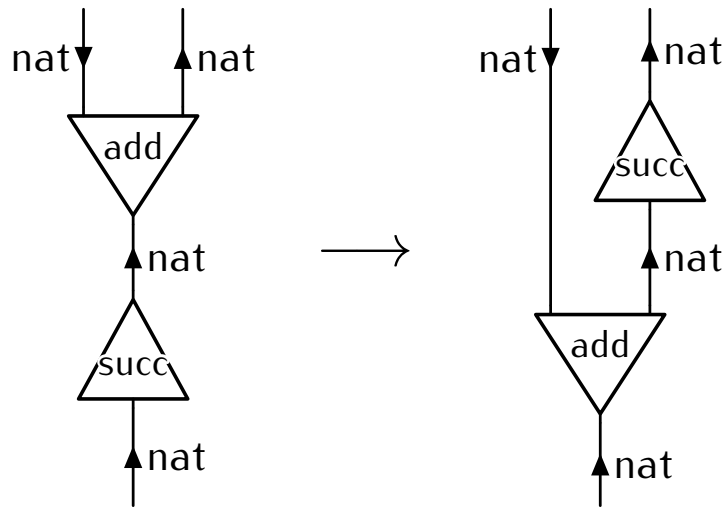
Typing:



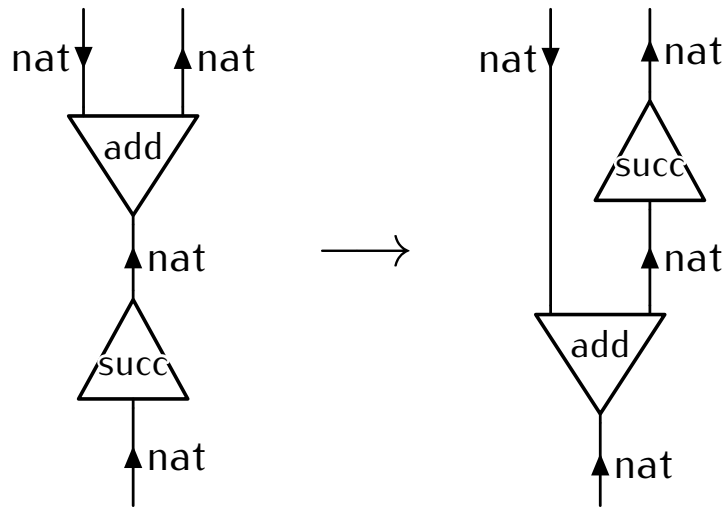
Type unification:



Reduction rules are required to be typed:



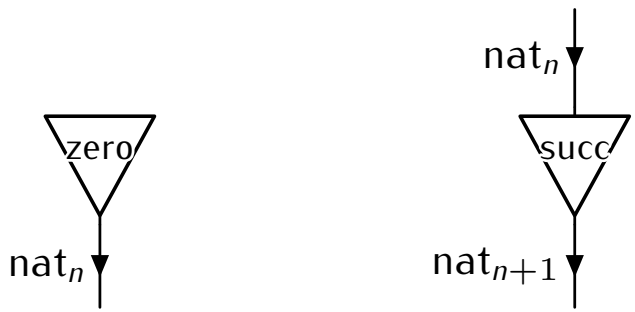
Reduction rules are required to be typed:



Interfaces of nets are preserved.

Sized Types

A type nat_n for natural numbers bounded by n:



Sized Types

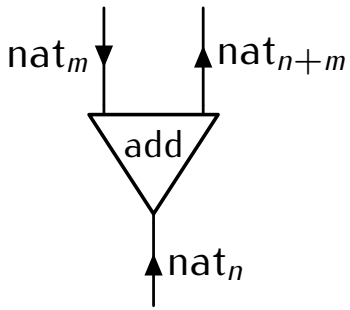
A type nat_n for natural numbers bounded by n:

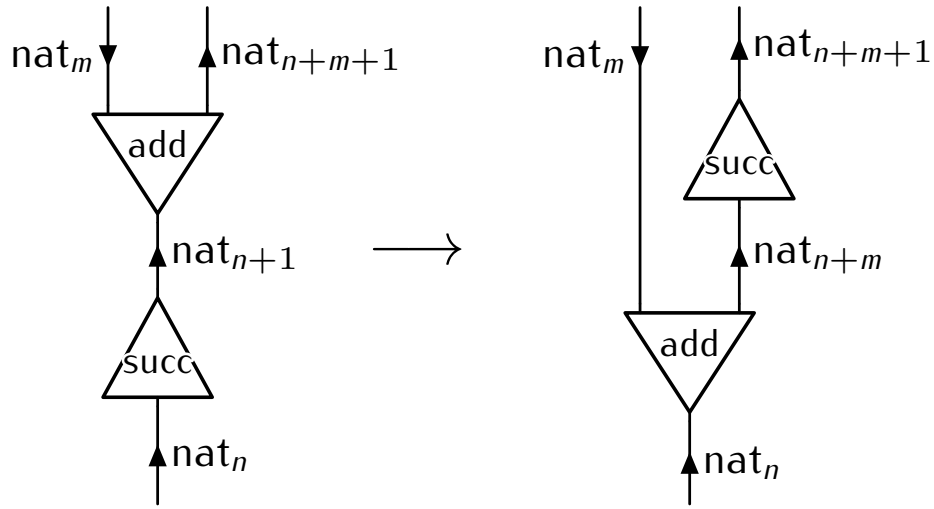
nat_n

nat_n nat_{n+1}

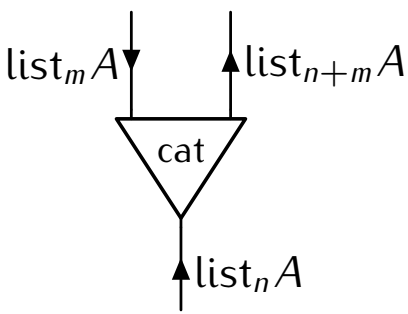
nat_n nat_m

Addition:





Concatenation:



Corollary (Sequential Time Complexity)

Associate $\tau_c \geq 0$, called time potential, to every typed node c, such that $\sum_{c \in L} \tau_c \geq \sum_{c \in R} \tau_c + d$ for every reduction rule $L \xrightarrow{d} R$.

$$N \xrightarrow{t} M \implies t \leq \sum_{c \in N} \tau_c$$

Corollary (Sequential Space Complexity)

Associate $\sigma_c \geq |c|$, called space potential, to every typed node c, such that $\sum_{c \in L} \sigma_c \geq \sum_{c \in R} \sigma_c$ for every reduction rule $L \xrightarrow{d} R$.

$$N \xrightarrow{t} M \implies |M| \leq \sum_{c \in N} \sigma_c$$

Theorem (Sequential Space–Time Complexity)

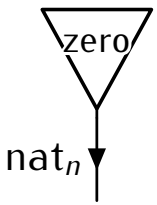
Associate a function $\gamma_c : T \to S$, called space–time potential, to every typed node c, such that $\gamma_c(0) \geq |c|$ and $(\coprod_{c \in L} \gamma_c)(t + d) \geq (\coprod_{c \in R} \gamma_c)(t)$ for every reduction rule $L \xrightarrow{d} R$.

$$N \xrightarrow{t} M \implies |M| \leq (\coprod_{c \in N} \gamma_c)(t)$$

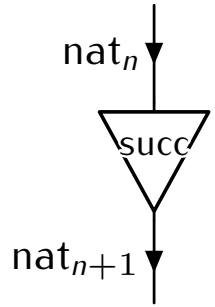
Proof.

The potential-decrease property assumed for reduction rules entails the same property for individual sequential reduction steps of a net. Hence, by combining these steps, we obtain $(\coprod_{c \in N} \gamma_c)(t) \geq \cdots \geq (\coprod_{c \in M} \gamma_c)(0) \geq |M|$. □

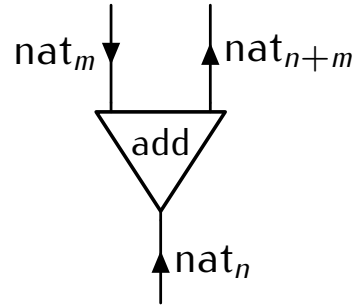
Assuming unitary weights, these potentials are compatible with reduction rules and therefore allow to compute sequential time and space bounds:



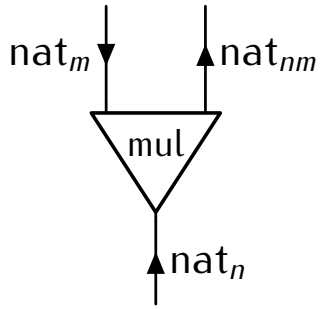
$\tau = 0$
$\sigma = 1$



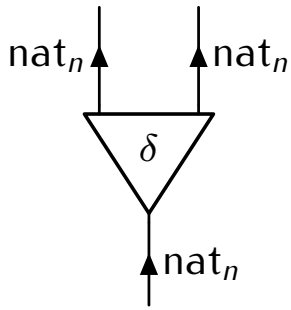
$\tau = 0$
$\sigma = 1$



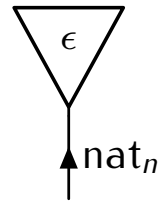
$\tau = n + 1$
$\sigma = 1$



$\tau = 2nm + 2n + m + 2$
$\sigma = nm + n + 1$



$\tau = n + 1$
$\sigma = n + 1$



$\tau = n + 1$
$\sigma = 1$

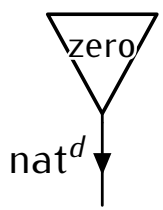
Scheduled Types

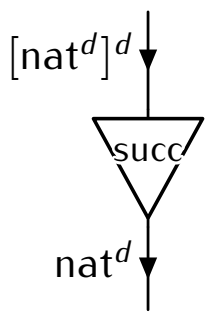
Delay Operator: An object of type $[A]^t$ will become an A after a parallel reduction of duration t.

Scheduled Types

Delay Operator: An object of type $[A]^t$ will become an A after a parallel reduction of duration t.

A type nat^d for natural numbers that are computed with pace d, i.e. the first constructor is available now and one new constructor will be made available after every parallel reduction of duration d (or faster):

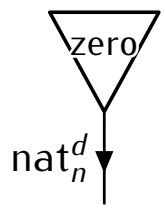


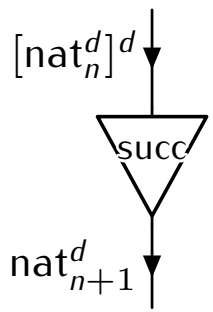


Scheduled Types

Delay Operator: An object of type $[A]^t$ will become an A after a parallel reduction of duration t.

A type nat^d for natural numbers that are computed with pace d, i.e. the first constructor is available now and one new constructor will be made available after every parallel reduction of duration d (or faster):

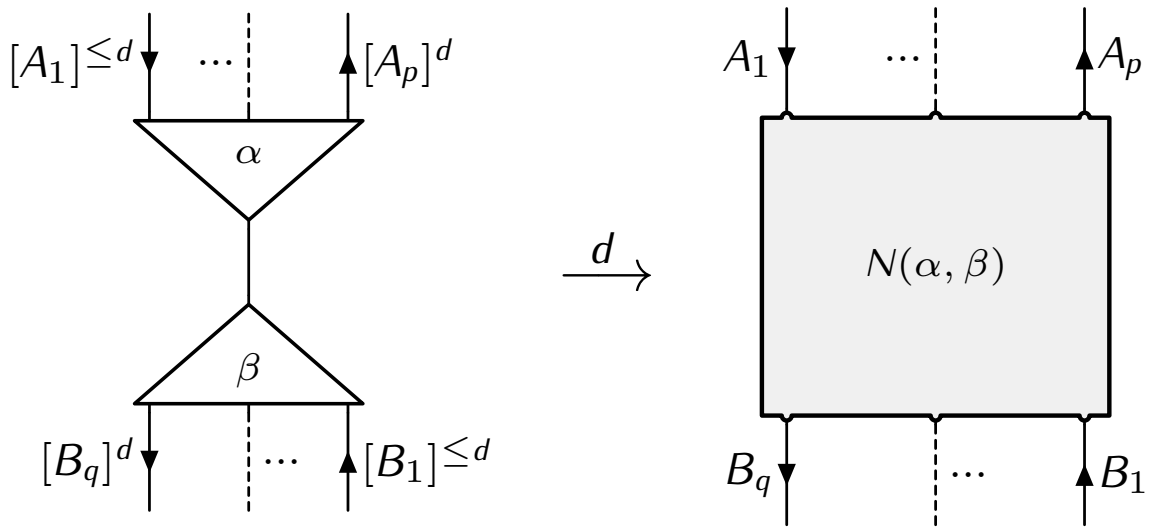




Timing annotations can be combined with size annotations.

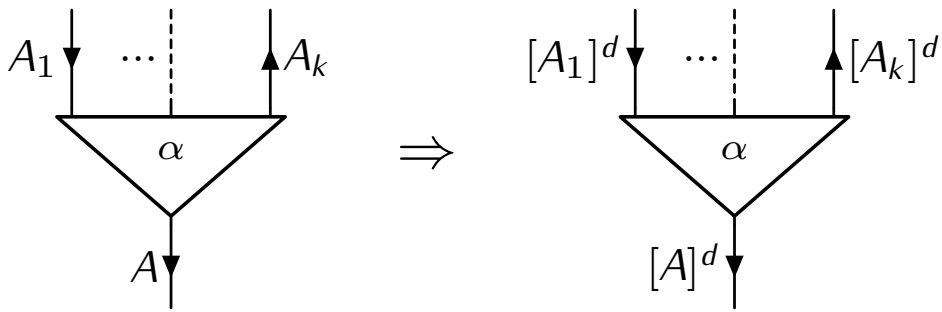
Reduction rules must meet the following requirement:

- Top-level output delays are reduced by the duration of the rule upon firing.
- Top-level input delays may, but need not, be reduced by the full amount of the duration of the rule.

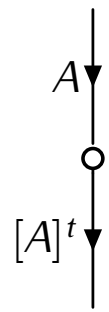


Two typing conveniences for scheduled types:

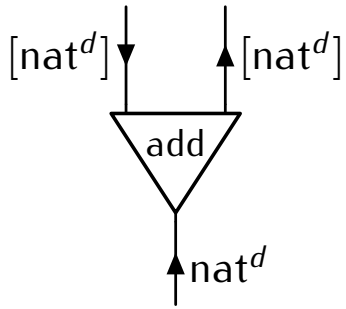
- *Delayed Computation*:



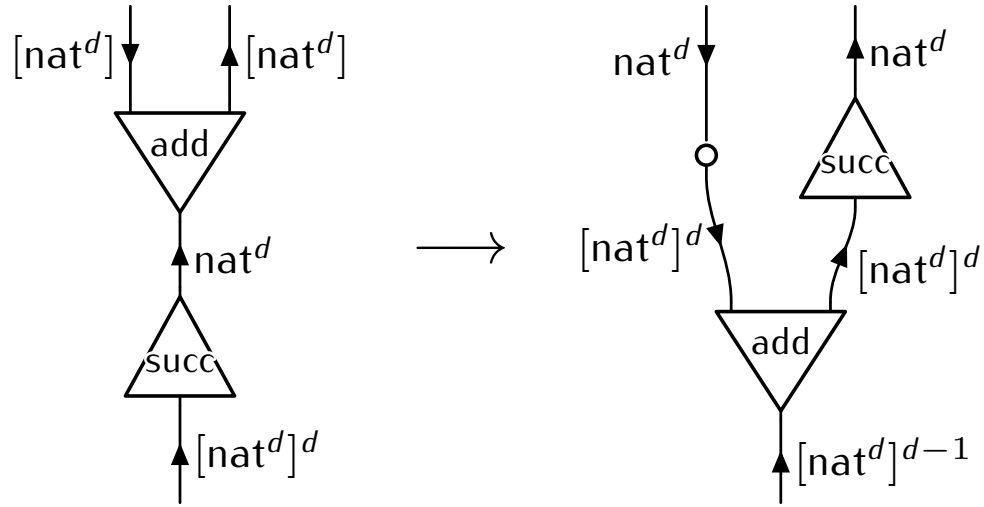
- *Subtyping*: When A is a tree-like data type:



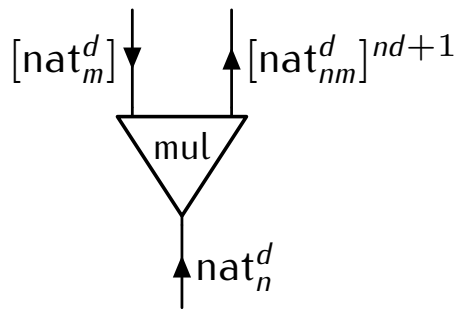
Addition:



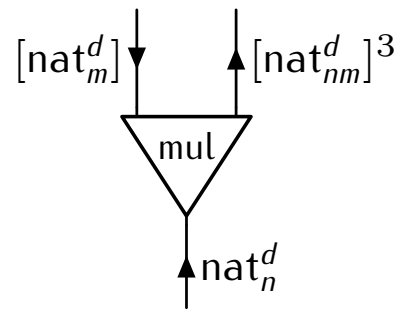
$d \geq 1$



Multiplication:



$d \geq 1$



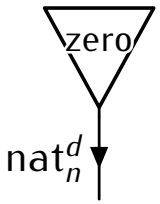
$d \geq 2$

Corollary (Parallel Time Complexity)

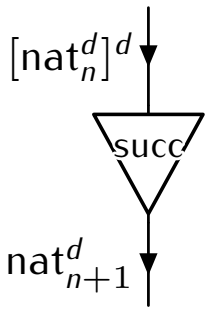
Associate $\bar{\tau}_c \geq 0$, called parallel time potential, to every schedule-typed node c, such that $\bar{\tau}_{[c]^d} \geq \bar{\tau}_c + d$ and $\max_{c \in L} \bar{\tau}_c \geq \max_{c \in R} \bar{\tau}_c + d$ for every reduction rule $L \xrightarrow{d} R$.

$$N \xrightarrow{t}_p M \implies t \leq \max_{c \in N} \bar{\tau}_c$$

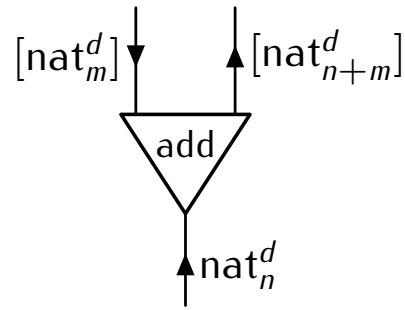
The following potentials are compatible with reduction rules and therefore provide parallel complexity measures:



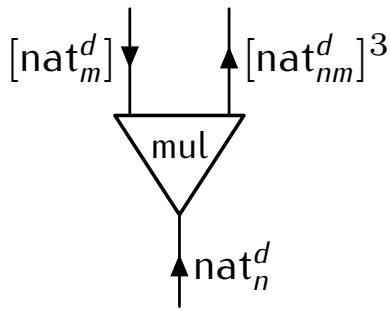
$\bar{\tau} = 0$
$\bar{\sigma} = 1$



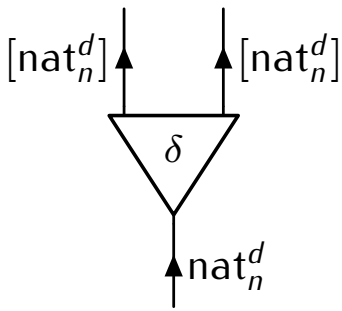
$\bar{\tau} = 0$
$\bar{\sigma} = 1$



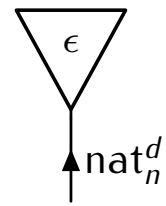
$\bar{\tau} = nd + 1$
$\bar{\sigma} = 1$



$\bar{\tau} = nd + md + 3$
$\bar{\sigma} = nm + 2n + 1$



$\bar{\tau} = nd + 1$
$\bar{\sigma} = n + 1$



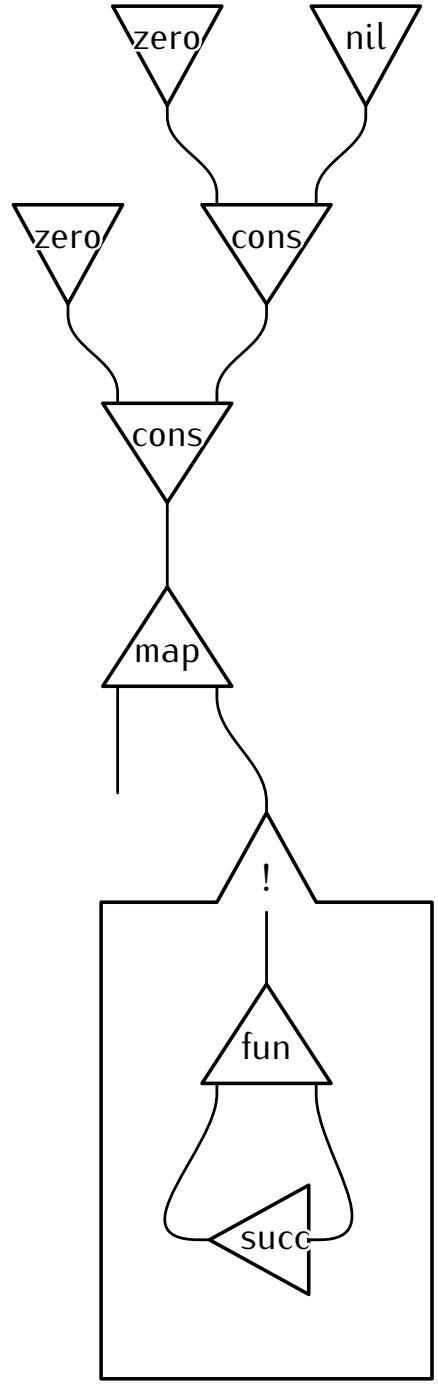
$\bar{\tau} = nd + 1$
$\bar{\sigma} = 1$

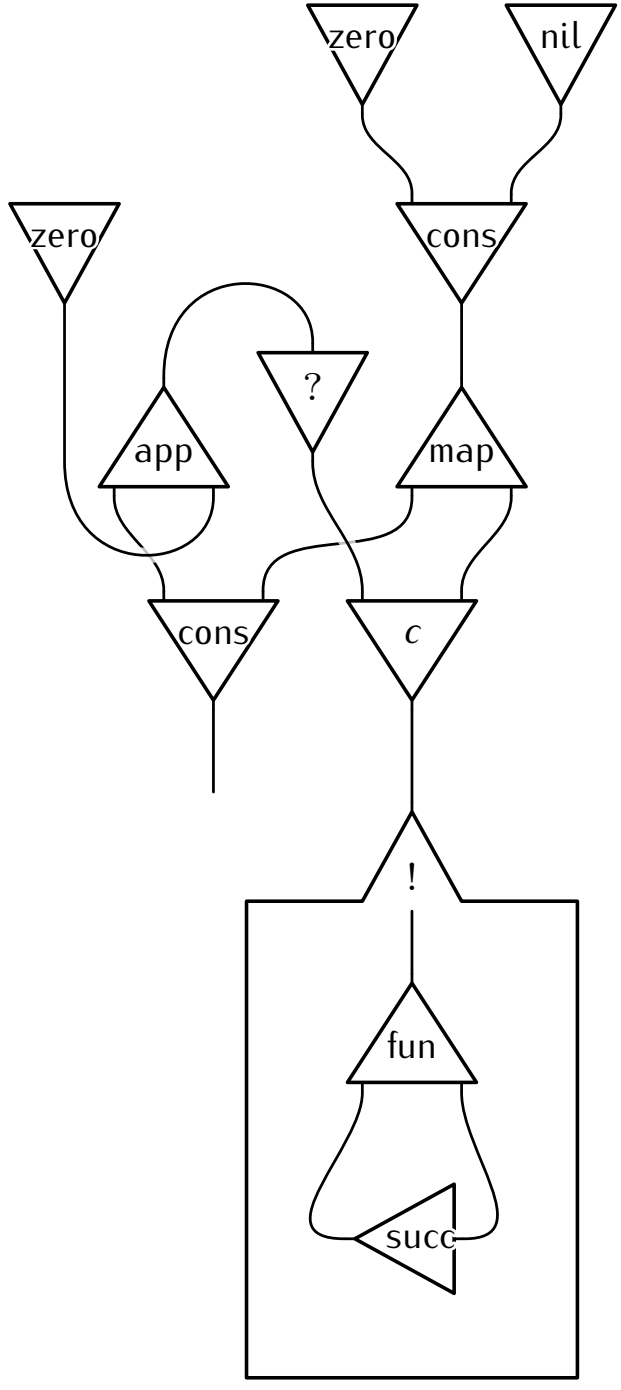
Higher-Order Resource Analysis

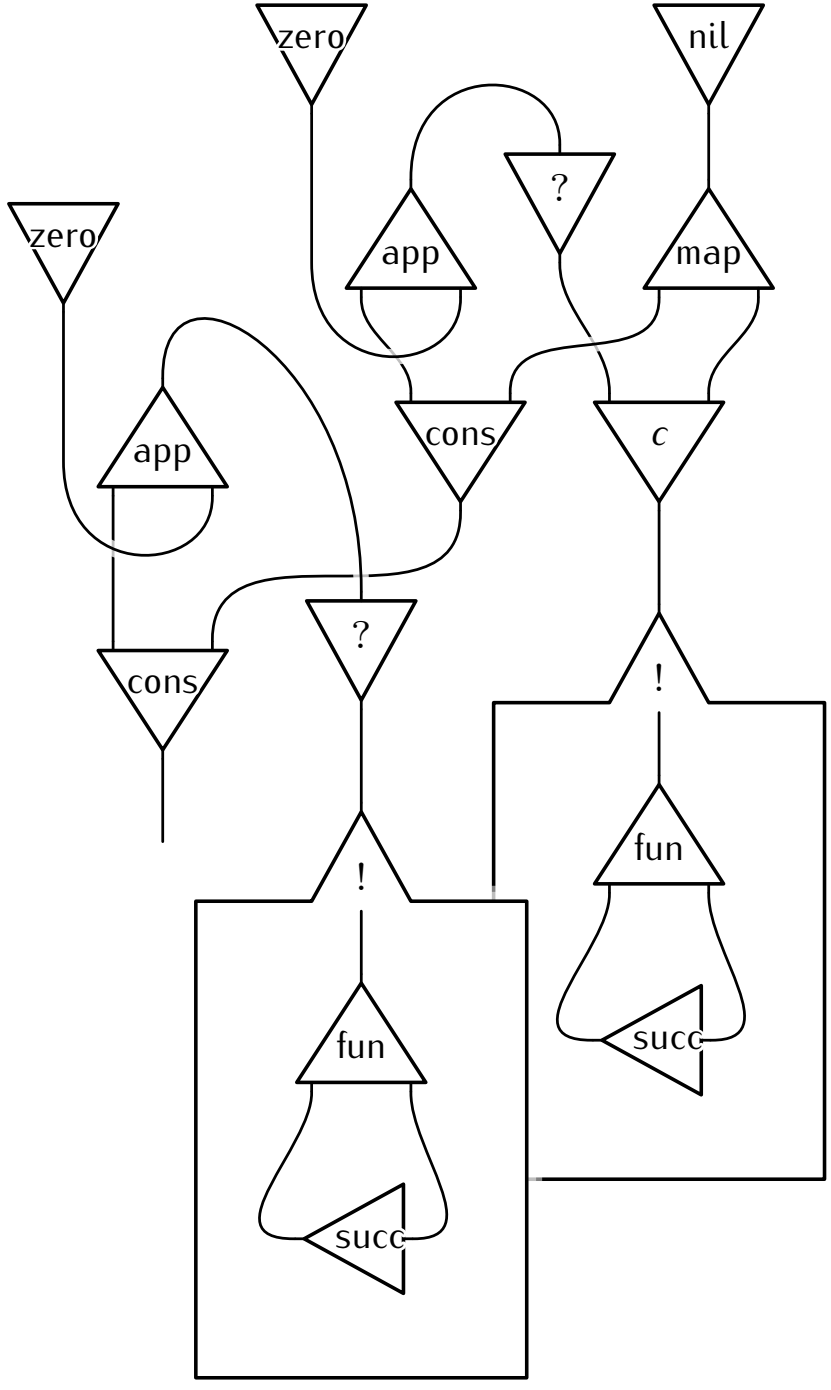
Some restrictions:

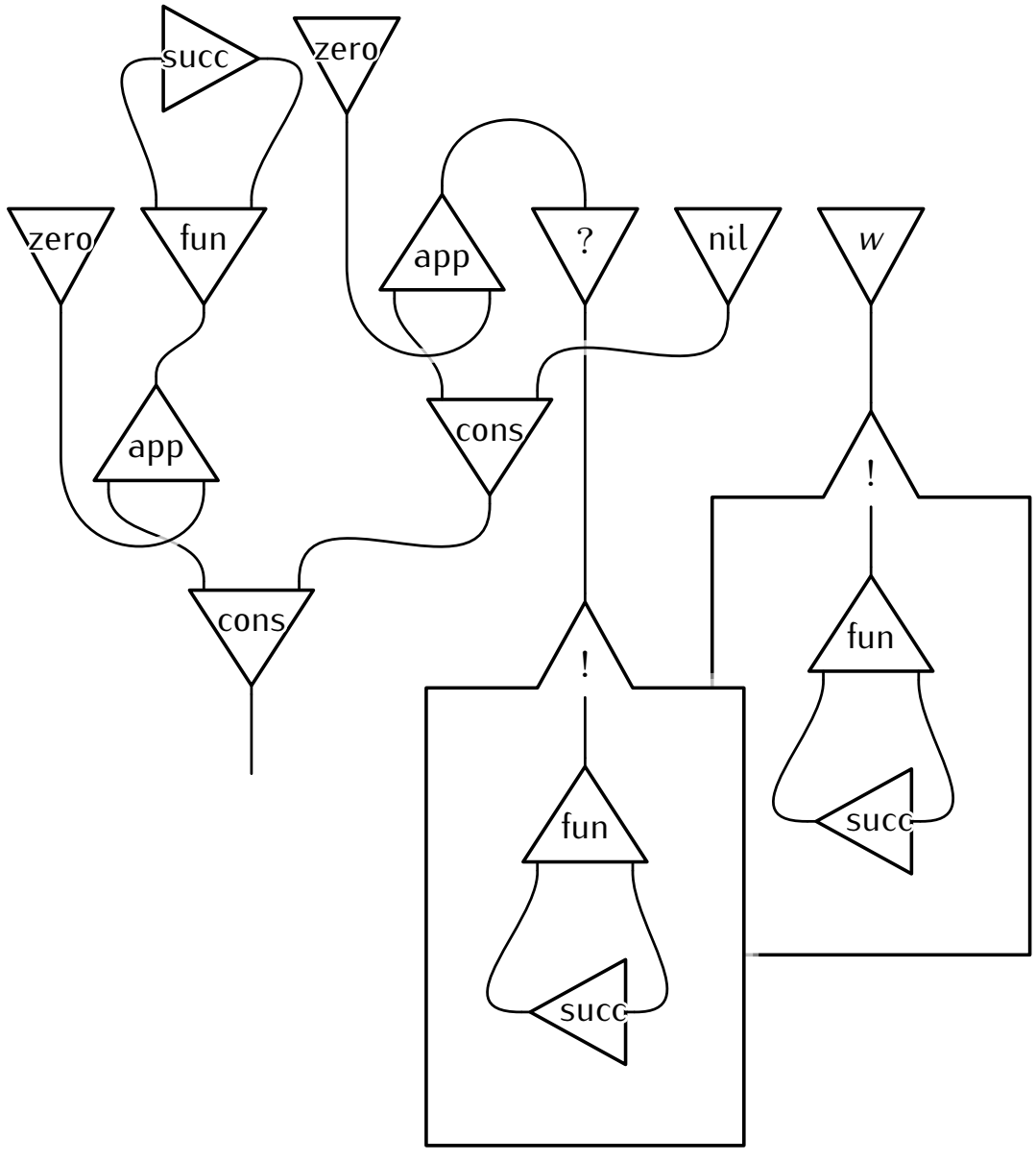
- Weak reduction (How to keep track of function sizes?)
- Sequential reduction ($A \trianglelefteq [A]^t$ unavailable for functional types)

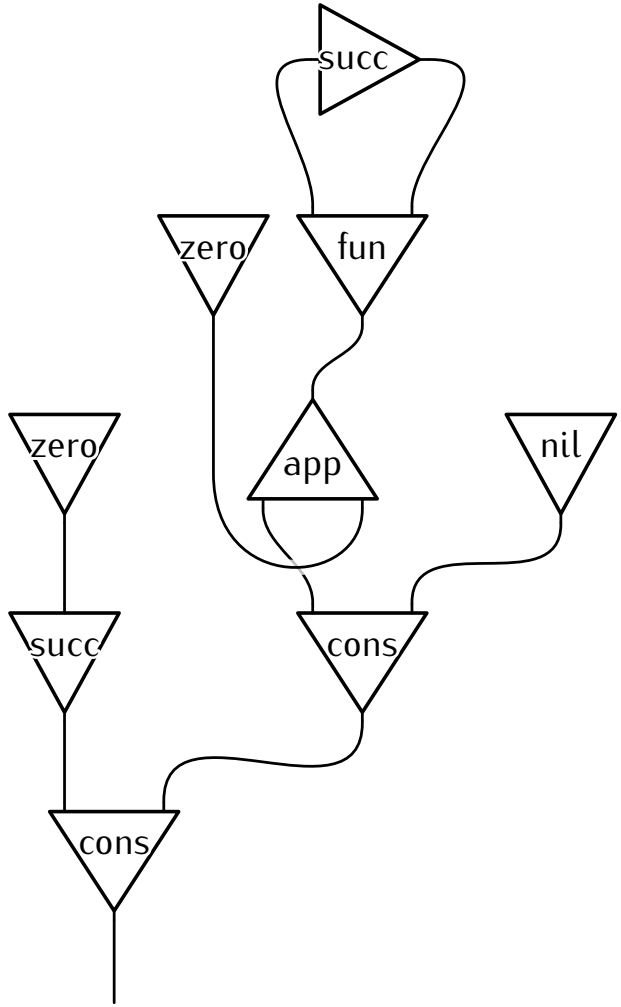
A model similar to top-level β-reduction, i.e. standard implementations of functional programming languages.

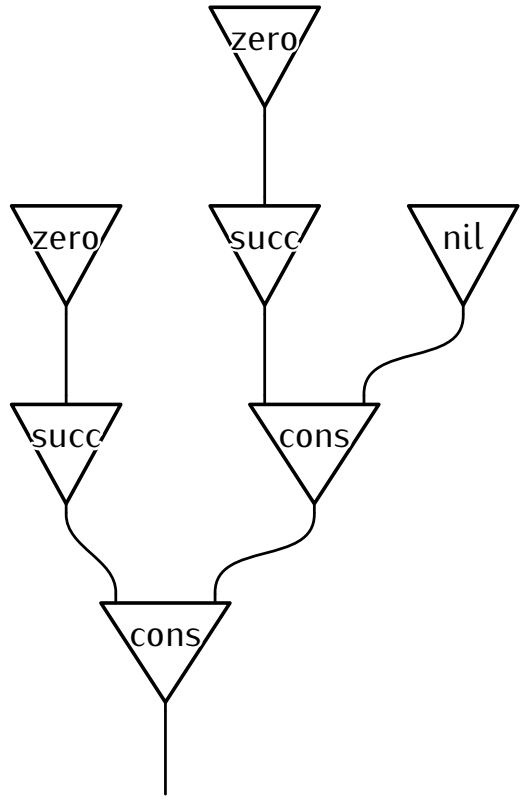

zero
nil
zero
cons
cons
map
!
fun
succ


zero
nil
zero
cons
?
app
map
cons
c
!
fun
succ

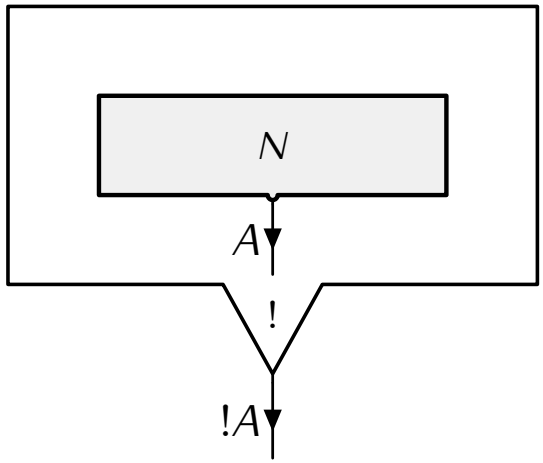

zero
nil
?
zero
app
map
app
cons
c
cons
?
!
fun
!
succ
fun
succ

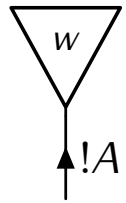

succ
zero
zero
fun
app
?
nil
w
app
cons
!
cons
fun
!
fun
succ
succ

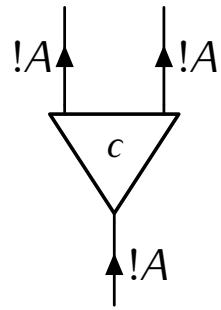

succ
zero
fun
zero
app
nil
succ
cons
cons

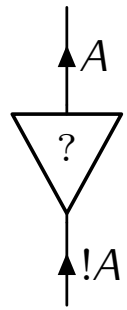

zero
zero
succ
nil
succ
cons
cons

Polymorphic Replication (aka Promotion):







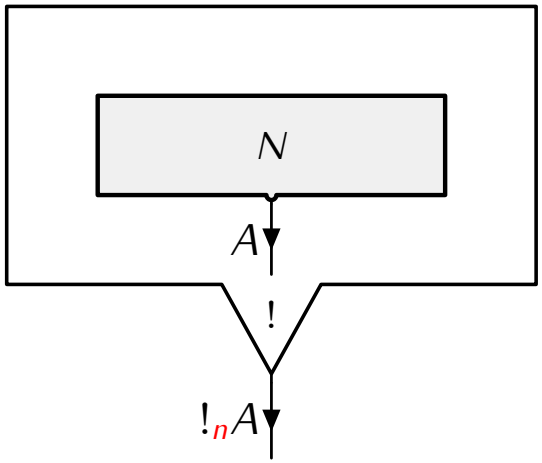


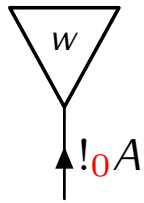
The sized version of polymorphicaly replicable types are simply multisets of (sized) types.

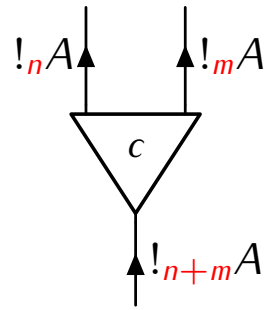
We use $!_{i<n}A_i$ as syntactic sugar to denote the multiset $\{A_0, \ldots, A_{n-1}\}$ and write $!_nA$ for homogeneous multisets that contain a single element with multiplicity n.

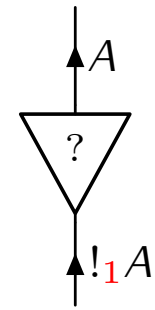
Multiset inclusion admissible as subtyping.

Polymorphic Replication (aka Promotion):







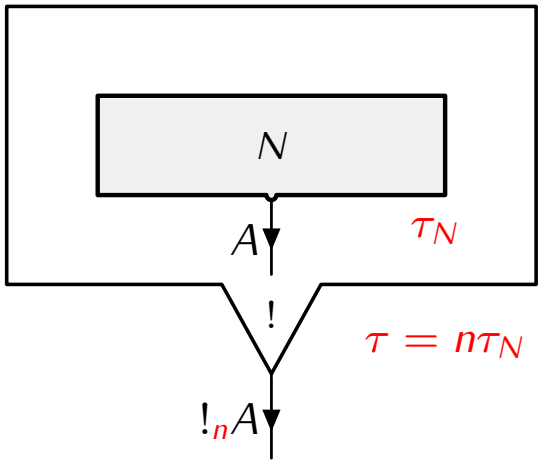


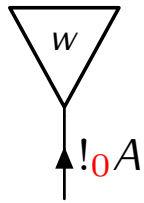
The sized version of polymorphicaly replicable types are simply multisets of (sized) types.

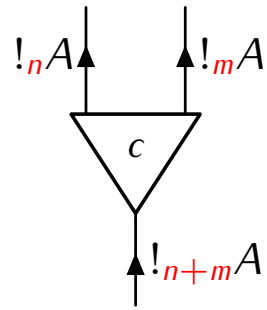
We use $!_{i<n}A_i$ as syntactic sugar to denote the multiset $\{A_0, \ldots, A_{n-1}\}$ and write $!_nA$ for homogeneous multisets that contain a single element with multiplicity n.

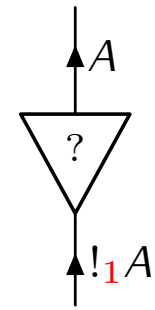
Multiset inclusion admissible as subtyping.

Polymorphic Replication (aka Promotion):







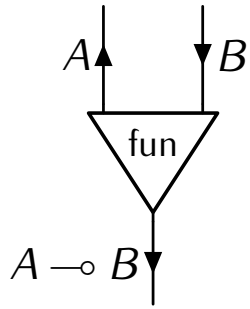


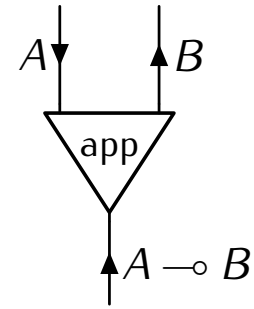
The sized version of polymorphicaly replicable types are simply multisets of (sized) types.

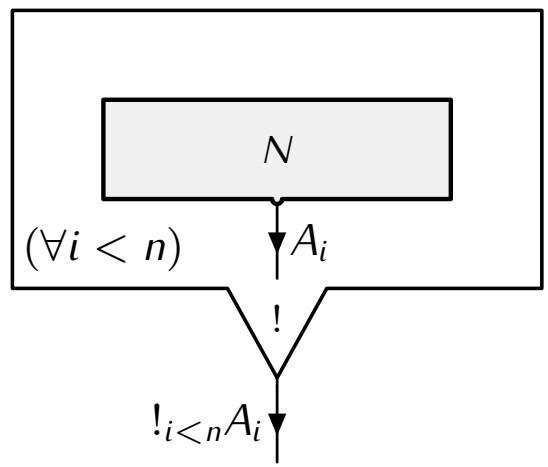
We use $!_{i<n} A_i$ as syntactic sugar to denote the multiset $\{A_0, \ldots, A_{n-1}\}$ and write $!_n A$ for homogeneous multisets that contain a single element with multiplicity n.

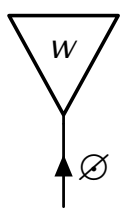
Multiset inclusion admissible as subtyping.

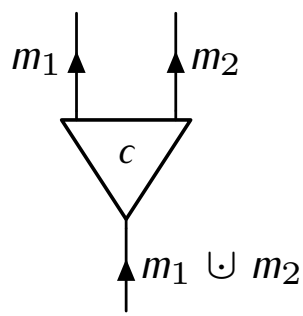
Types and potentials can be attributed as follows:

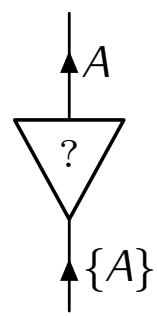




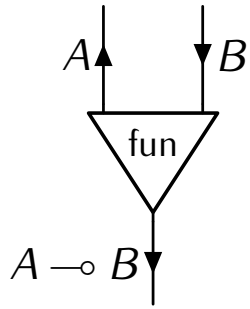




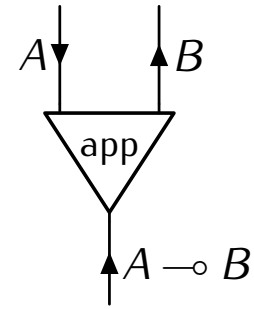




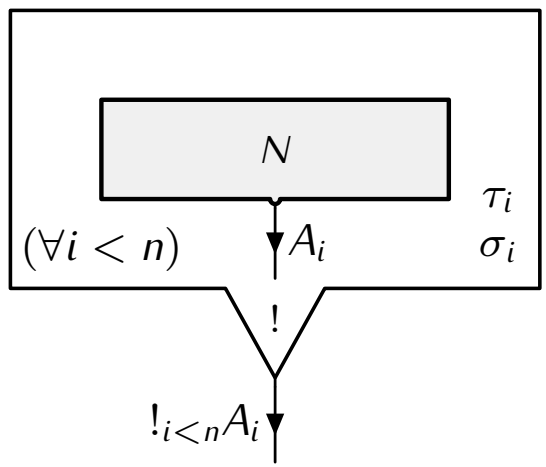
Types and potentials can be attributed as follows:



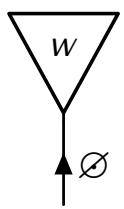
$\tau = 0$
$\sigma = 1$



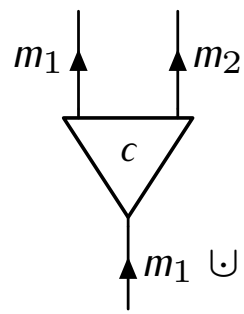
$\tau = 1$
$\sigma = 1$



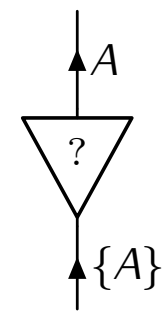
$\tau = \sum_{i<n} \tau_i$
$\sigma = 1 + \sum_{i<n} \sigma_i$



$\tau = 1$
$\sigma = 1$

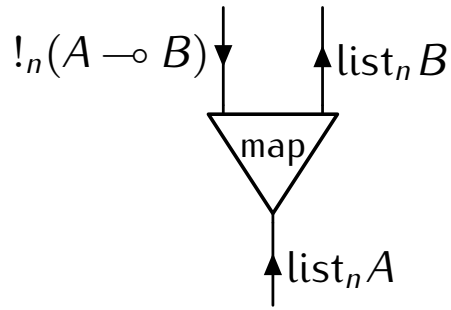


$\tau = 1$
$\sigma = 1$

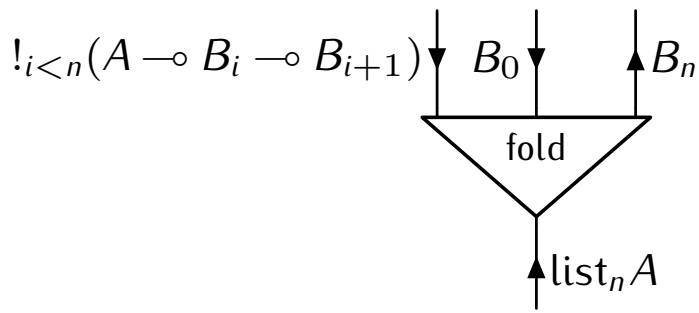


$\tau = 1$
$\sigma = 1$

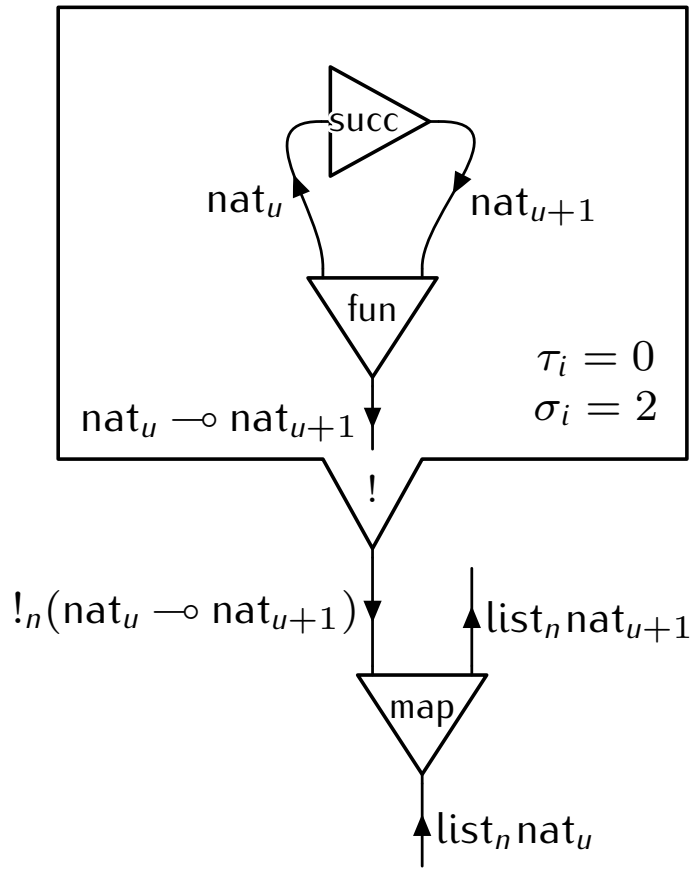
Map:


$!_n(A \multimap B)$
$\mathsf{list}_n B$
map
$\mathsf{list}_n A$

Fold:


$!_{i<n}(A \multimap B_i \multimap B_{i+1})$
B_0
B_n
fold
$\mathsf{list}_n A$

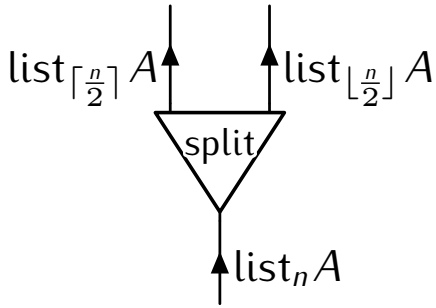
Resource usage of functional arguments to map (or fold) are taken into account as the potential of the box which holds them.

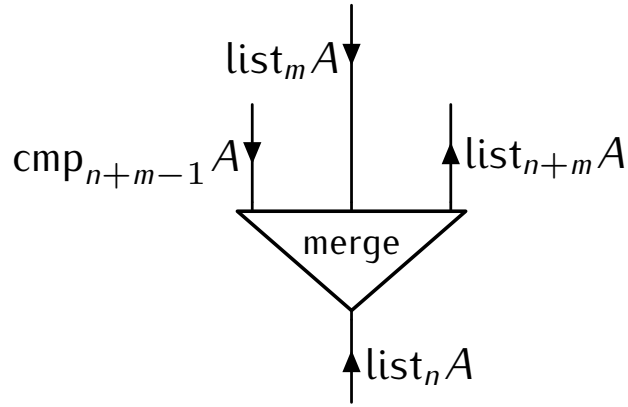


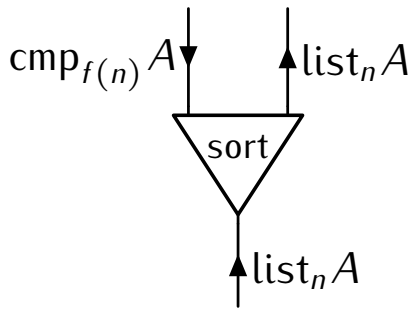
This defines an operation of type $\mathsf{list}_n\,\mathsf{nat}_u \multimap \mathsf{list}_n\,\mathsf{nat}_{u+1}$ with:

- A combined $\tau = 2 + 4n + \sum_{i<n} 0 = 2 + 4n$
- A combined $\sigma = 1 + 4n + 1 + \sum_{i<n} 2 = 2 + 6n$

Merge sort:

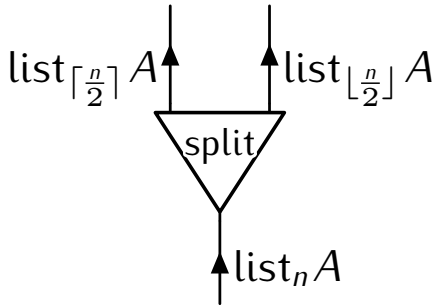


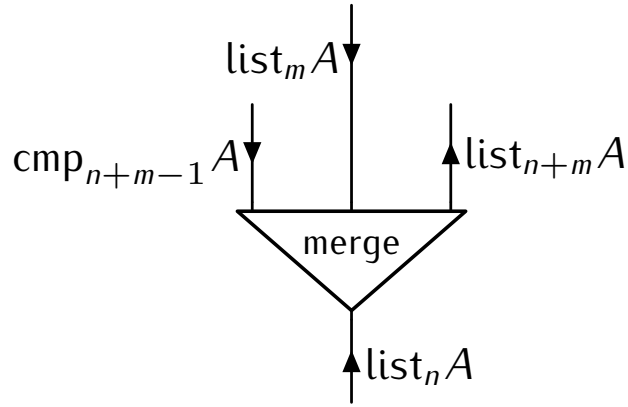


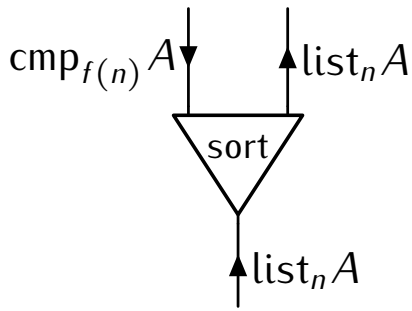


$$
\begin{aligned}
\mathrm{cmp}_n A &= !_n(A \multimap A \multimap \mathrm{bool}) \\
f(n) &= n - 1 + f(\lfloor \frac{n}{2} \rfloor) + f(\lceil \frac{n}{2} \rceil) \\
&\leq n \log_2 n
\end{aligned}
$$

Merge sort:



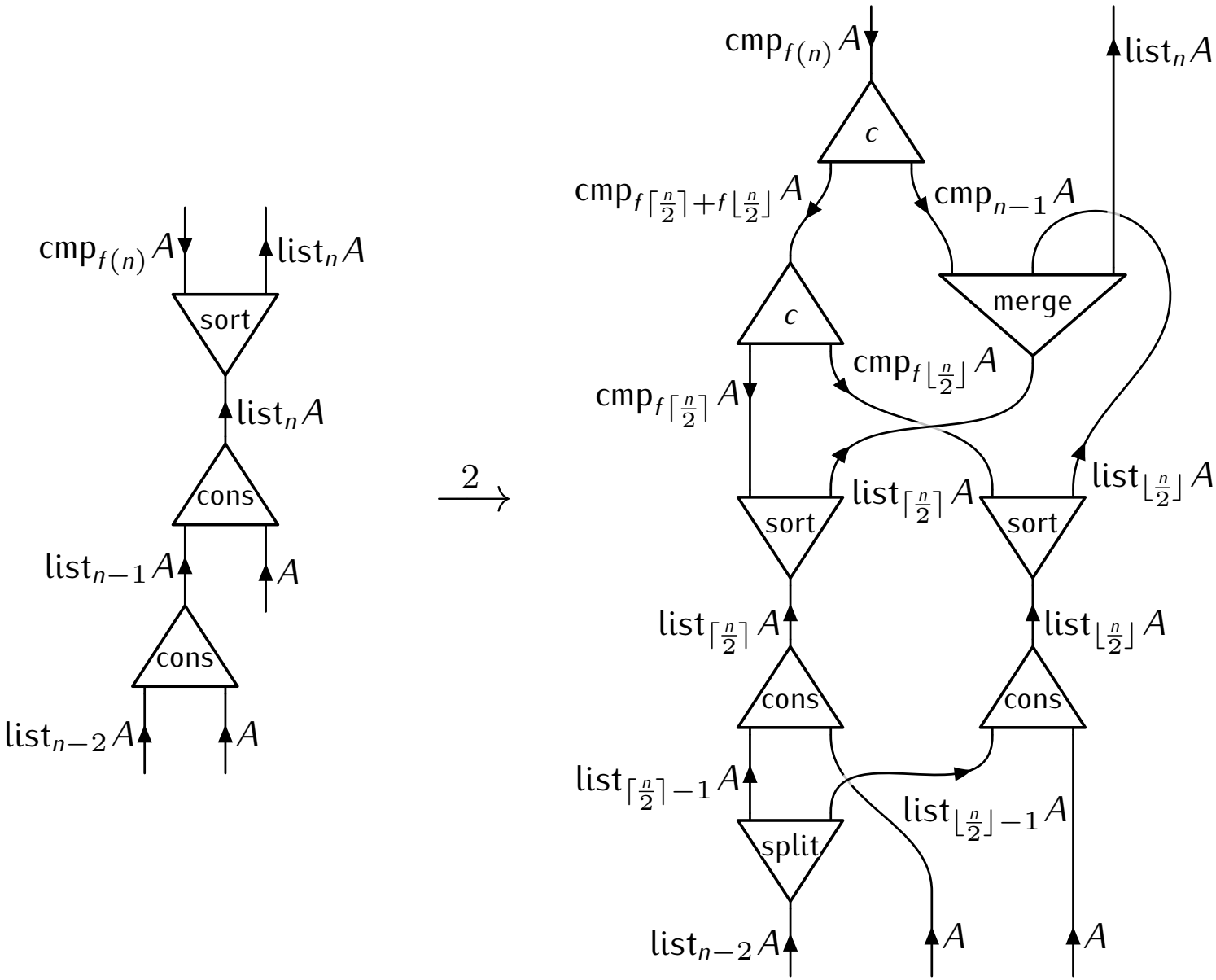




$$\tau = 4f(n) + 10n - 8$$
$$\sigma = 8n - 7$$

$$\begin{aligned} \mathrm{cmp}_n A &= !_n(A \multimap A \multimap \mathrm{bool}) \\ f(n) &= n - 1 + f(\lfloor \frac{n}{2} \rfloor) + f(\lceil \frac{n}{2} \rceil) \\ &\leq n \log_2 n \end{aligned}$$

Main reduction path:



Conclusion

A methodology for checkable complexity specifications.

- different resources: space, time, space–time
- sequential and parallel models
- type-based (sized types, scheduled types)
- potential-based (amortized analysis)
- general: logarithmic (*msort*) or exponential (*fold mult*) complexities
- higher-order (with some restrictions)

Future work:

- partial automation

Thanks! Questions?