Resource Constrained Programming with Full Dependent Types

Robert Atkey
Strathclyde University, Glasgow
robert.atkey@strath.ac.uk

CCS Colloquium, Augusta University
20th November 2020

Dependent Type Theory is both

- Programming Language
- Proof Language

Dependent Type Theory is both

- Programming Language
- Proof Language

So we can write programs

Dependent Type Theory is both

- Programming Language
- Proof Language

So we can write programs

and reason about them

Dependent Type Theory is both

- Programming Language

So we can write programs

- Proof Language

and reason about them
but only the “extensional behaviour”

What if we want to reason about computational complexity?

What if we want to reason about computational complexity?

Having predicates for complexity won't work:

$$\mathrm{Ptime} : (\mathrm{Nat} \to \mathrm{Nat}) \to \mathrm{Set}$$

Allows the theory to distinguish extensionally equivalent functions.

What if we want to reason about computational complexity?

What if we want to reason about computational complexity?

Two ideas:

- Implicit: all functions are in a fixed complexity class (e.g., PTIME)
- Explicit: types tell us what the complexity is.

This talk

- Implicit and explicit typed complexity analysis for Dependent Type Theory

Challenges

- Nice systems for implicit and explicit complexity
- Integrating them with dependent types

Two Implicit PTIME systems

Requirements

- Extension of typed λ-calculus; *higher order*
- No impredicative polymorphism (no Church encodings)
- Proper datatypes (definitely no Church encodings)

Requirements

- Extension of typed λ-calculus; *higher order*
- No impredicative polymorphism (no Church encodings)
- Proper datatypes (definitely no Church encodings)

Forget dependent types for now

- Simply typed λ-calculus
- A natural number type $\textsc{Nat}$, zero, suc with an iterator

$$\frac{\Gamma \vdash M_z : A \qquad \Gamma, x : A \vdash M_s : A \qquad \Gamma \vdash N : \textsc{Nat}}{\Gamma \vdash \mathrm{iter}(M_z, x.\, M_s, N) : A}$$

Easily yields exponential time:

$$\text{iter}(\text{suc}, f.\ \lambda x.\ f(f(x)), N)\ \text{zero} : \textsc{Nat}$$

computes 2^N

Easily yields exponential time:

$$\text{iter}(\text{suc}, f.\ \lambda x.\ f(f(x)), N)\ \text{zero} : \textsc{Nat}$$

computes 2^N

Culprits

- Duplication of the higher order value f
- Construction of new numbers

Linearity?

Disallows:

$$\text{iter}(\text{suc}, f.\ \lambda x.\ f(f(x)), N)\ \text{zero} : \textsc{Nat}$$

because f is used twice.

Linearity?

Disallows:

$$\text{iter}(\text{suc}, f.\ \lambda x.\ f(f(x)), N)\ \text{zero} : \textsc{Nat}$$

because f is used twice.

But

Can write:

$$\begin{aligned} \text{dup} \quad &: \quad \textsc{Nat} \multimap \textsc{Nat} \otimes \textsc{Nat} \\ \text{dup}\ x \quad &= \quad \text{iter}((\text{zero}, \text{zero}), (m, n).(\text{suc}\ m, \text{suc}\ n), x) \end{aligned}$$

Linearity?

Disallows:

$$\mathrm{iter}(\mathrm{suc}, f.\ \lambda x.\ f(f(x)), N)\ \mathrm{zero} : \textsc{Nat}$$

because f is used twice.

But

Can write:

$$\begin{aligned} \mathrm{dup} &: \textsc{Nat} \multimap \textsc{Nat} \otimes \textsc{Nat} \\ \mathrm{dup}\ x &= \mathrm{iter}((\mathrm{zero}, \mathrm{zero}), (m, n).(\mathrm{suc}\ m, \mathrm{suc}\ n), x) \end{aligned}$$

- add : $\textsc{Nat} \multimap \textsc{Nat} \multimap \textsc{Nat}$ is linear.

Linearity?

Disallows:

$$\text{iter}(\text{suc}, f.\ \lambda x.\ f(f(x)), N)\ \text{zero} : \textsc{Nat}$$

because f is used twice.

But

Can write:

$$\begin{array}{lcl} \text{dup} & : & \textsc{Nat} \multimap \textsc{Nat} \otimes \textsc{Nat} \\ \text{dup}\ x & = & \text{iter}((\text{zero}, \text{zero}), (m, n).(\text{suc}\ m, \text{suc}\ n), x) \end{array}$$

- add : $\textsc{Nat} \multimap \textsc{Nat} \multimap \textsc{Nat}$ is linear.
- mul : $\textsc{Nat} \multimap \textsc{Nat} \multimap \textsc{Nat}$ can be written using dup, add.

Linearity?

Disallows:

$$\mathrm{iter}(\mathrm{suc}, f.\ \lambda x.\ f(f(x)), N)\ \mathrm{zero} : \textsc{Nat}$$

because f is used twice.

But

Can write:

$$\begin{array}{lcl} \mathrm{dup} & : & \textsc{Nat} \multimap \textsc{Nat} \otimes \textsc{Nat} \\ \mathrm{dup}\ x & = & \mathrm{iter}((\mathrm{zero}, \mathrm{zero}), (m, n).(\mathrm{suc}\ m, \mathrm{suc}\ n), x) \end{array}$$

- add : $\textsc{Nat} \multimap \textsc{Nat} \multimap \textsc{Nat}$ is linear.
- mul : $\textsc{Nat} \multimap \textsc{Nat} \multimap \textsc{Nat}$ can be written using dup, add.
- exp : $\textsc{Nat} \multimap \textsc{Nat} \multimap \textsc{Nat}$ can be written using dup, mul.

Linearity?

Disallows:

$$\text{iter}(\text{suc}, f.\ \lambda x.\ f(f(x)), N)\ \text{zero} : \textsc{Nat}$$

because f is used twice.

But

Can write:

$$\begin{array}{lcl} \text{dup} & : & \textsc{Nat} \multimap \textsc{Nat} \otimes \textsc{Nat} \\ \text{dup}\ x & = & \text{iter}((\text{zero}, \text{zero}), (m, n).(\text{suc}\ m, \text{suc}\ n), x) \end{array}$$

- add : $\textsc{Nat} \multimap \textsc{Nat} \multimap \textsc{Nat}$ is linear.
- mul : $\textsc{Nat} \multimap \textsc{Nat} \multimap \textsc{Nat}$ can be written using dup, add.
- exp : $\textsc{Nat} \multimap \textsc{Nat} \multimap \textsc{Nat}$ can be written using dup, mul.
- Get exponential time.

Linearity + No constructors

- Can't write dup or add (or mul or exp)

Linearity + No constructors

- Can't write dup or add (or mul or exp)

- Iterable NAT:
  - Not constructible
  - Has an iterator

Linearity + No constructors

- Can't write dup or add (or mul or exp)

- Iterable Nat:
  - Not constructible
  - Has an iterator

- Non-iterable Nat°:
  - Constructible
  - Case analysis

$$\frac{\Gamma_1 \vdash M_z : A \qquad \Gamma_2, x : \text{Nat}^\circ \vdash M_s : A \qquad \Gamma_3 \vdash N : \text{Nat}^\circ}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \text{case}(M_z, x.\ M_s, N) : A}$$

Is this enough?

- Only source of iterable Nat is the input
- So only linear time in the size of the Nat “fuel” provided
- To get polytime, allow duplication of variables of type Nat.

Is this enough?

- Only source of iterable NAT is the input
- So only linear time in the size of the NAT "fuel" provided
- To get polytime, allow duplication of variables of type NAT.

Completeness

- Given a step function $s : \text{TAPE} \multimap \text{TAPE}$, and a $\mathbb{N}$-polynomial $p(n) = \Sigma a_i n^i$
- n iterations: $\mathrm{iter}(\lambda x.x, f.\lambda x.\ s(f\,x), n) : \text{TAPE} \multimap \text{TAPE}$
- n^2 iterations: $\mathrm{iter}(\lambda x.x, f.\lambda x.\ \mathrm{iter}(\lambda x.x, f.\lambda x.\ s(f\,x), n), n) : \text{TAPE} \multimap \text{TAPE}$
- n^i iterations...
- Addition by composition

Recovering Constructibility?

- This system works, but is restricted to everything being driven by Nat-iteration
- Some programs are more easily expressible by iteration over trees, etc.

Martin Hofmann's LFPL: principle of "conservation of iterability"

- A special type $\diamond$, representing a chunk of iterability
- Required for construction:

$$\mathrm{zero} : \diamond \multimap \textsc{Nat} \qquad\qquad \mathrm{suc} : \diamond \multimap \textsc{Nat} \multimap \textsc{Nat}$$

- Recovered on iteration:

$$\frac{\Gamma_1, d : \diamond \vdash M_z : A \qquad d : \diamond, x : A \vdash M_s : A \qquad \Gamma_2 \vdash N : \textsc{Nat}}{\Gamma_1, \Gamma_2 \vdash \mathrm{iter}(d.\ M_z, d\ x.\ M_s, N) : A}$$

- Extends easily to other datatypes

Iterating a step function

- Assume we have a function $s : \textsc{Tape} \multimap \textsc{Tape}$

  one step of a Turing machine

- Linear $\binom{n}{1}$ iterations:

$$\begin{aligned} I_1 = \lambda(n, t).\mathrm{iter}(&d.\ (\mathrm{zero}(d), t), \\ &d\ (n, t).\ (\mathrm{suc}(d, n), s\ t), \\ &n) \end{aligned} \quad : \textsc{Nat} \otimes \textsc{Tape} \multimap \textsc{Nat} \otimes \textsc{Tape}$$

- $\binom{n}{2}$ iterations:

$$\begin{aligned} I_2 = \lambda(n, t).\mathrm{iter}(&d.\ (\mathrm{zero}(d), t), \\ &d\ (n, t).\ \mathrm{let}\ (n, t) = I_1(n, t)\ \mathrm{in}\ (\mathrm{suc}(d, n), s(t)), \\ &n) \end{aligned} \quad : \textsc{Nat} \otimes \textsc{Tape} \multimap \textsc{Nat} \otimes \textsc{Tape}$$

- $\binom{n}{3}$ iterations: Iterate the above

Iterating a step function

- Obtain a $\binom{n}{k}$ iterator for any k
- And get the original number back as an output
- Chain them together to get any polynomial:

$$p(n) = \sum_{i=0}^{k} p_i \binom{n}{k}$$

- So we get polytime completeness

Explicit Complexity

Amortised Resource Analysis — (Hofmann & Jost, POPL 2003)

- Reinterpret $\diamond$ as the cost of a step of iteration
- Inspired by Tarjan's *amortised complexity analyis*
  - storing potential inside data structures
- Building a $\textsc{Nat}$ still requires $\diamond$s:

$$\mathrm{zero} : \diamond \multimap \textsc{Nat} \qquad\qquad \mathrm{suc} : \diamond \multimap \textsc{Nat} \multimap \textsc{Nat}$$

- But iteration no longer gives you them back:

$$\frac{\Gamma_1 \vdash M_z : A \qquad x : A \vdash M_s : A \qquad \Gamma_2 \vdash N : \textsc{Nat}}{\Gamma_1, \Gamma_2 \vdash \mathrm{iter}_A(M_z, x.M_s, N) : A}$$

- Back to linear time...

More flexibility

- Annotate data structures with number of $\diamond$s per constructor

$$\textsc{Nat}^{p}$$

- Duplication:

$$\textsc{Nat}^{p_1+p_2} \multimap \textsc{Nat}^{p_1} \otimes \textsc{Nat}^{p_2}$$

- Hofmann & Jost (2001) used linear programming to infer the ps

Regaining polynomial time — (Hoffmann & Hofmann, ESOP 2010)

- Annotate with sequences of naturals:

$$\textsc{Nat}^{(p_1,\ldots,p_k)}$$

- Interpretation is that

$$\sum_{i=1}^{k} p_i \binom{n}{i}$$

is the number of $\diamond$s is attached to a natural n.

Regaining polynomial time — (Hoffmann & Hofmann, ESOP 2010)

- Annotate with sequences of naturals:

$$\mathrm{N{\small AT}}^{(p_1,\ldots,p_k)}$$

- Interpretation is that

$$\sum_{i=1}^{k} p_i \binom{n}{i}$$

is the number of $\diamond$s is attached to a natural n.

- Iterator:

$$\frac{\Gamma_1 \vdash M_z : A \qquad n : \mathrm{N{\small AT}}^{(p_1+p_2, p_2+p_3, \ldots, p_k)}, d : \diamond^{p_1}, x : A \vdash M_s : A \qquad \Gamma_2 \vdash N : \mathrm{N{\small AT}}^{(p_1+1, \ldots, p_k)}}{\Gamma_1, \Gamma_2 \vdash \mathrm{iter}(M_z, n\ d\ x.M_s, N) : A}$$

Adapting these systems to dependent types

Dependency and Accountancy

In Martin-Löf Type Theory

$$x_1 : S_1, \ldots, x_n : S_n \vdash M : T$$

In Martin-Löf Type Theory

$$x_1 : S_1, \ldots, x_n : S_n \vdash M : T$$

variables $x_1, \ldots, x_n$ are mixed usage

$$n : Nat, x : \mathsf{Fin}(n) \vdash x : \mathsf{Fin}(n)$$

$$n : Nat, x : \mathsf{Fin}(n) \vdash x : \mathsf{Fin}(n)$$

x is used *computationally*

$$n : Nat, x : \mathsf{Fin}(n) \vdash x : \mathsf{Fin}(n)$$

x is used *computationally*

n is used *logically*

In Linear Logic

$$x_1 : X_1, \ldots, x_n : X_n \vdash M : Y$$

In Linear Logic

$$x_1 : X_1, \ldots, x_n : X_n \vdash M : Y$$

the presence of a variable x records its usage
each x_i must be "used" by M exactly once

In Linear Logic

$$x_1 : X_1, \ldots, x_n : X_n \vdash M : Y$$

the presence of a variable x records its usage
each x_i must be "used" by M exactly once

Enables:

1. Insight into computational behaviour
2. e.g., time complexity

$$n : \mathsf{Nat}, x : \mathsf{Fin}(n) \vdash x : \mathsf{Fin}(n)$$

Can we read this judgement linearly?

$$n : \mathsf{Nat}, x : \mathsf{Fin}(n) \vdash x : \mathsf{Fin}(n)$$

Can we read this judgement linearly?

- n appears in the context, but is not used computationally

$$n : \mathsf{Nat}, x : \mathsf{Fin}(n) \vdash x : \mathsf{Fin}(n)$$

Can we read this judgement linearly?

- ▷ n appears in the context, but is not used computationally
- ▷ n appears *twice* in types

$$n : \mathsf{Nat}, x : \mathsf{Fin}(n) \vdash x : \mathsf{Fin}(n)$$

Can we read this judgement linearly?

▷ n appears in the context, but is not used computationally

▷ n appears *twice* in types

Is n even used at all?

$$n : \mathsf{Nat} \mid x : \mathsf{Fin}(n) \vdash x : \mathsf{Fin}(n)$$

$$n : \mathsf{Nat} \mid x : \mathsf{Fin}(n) \vdash x : \mathsf{Fin}(n)$$

- Separate *intuitionistic* / *unrestricted* uses and *linear* uses

$$n : \mathsf{Nat} \mid x : \mathsf{Fin}(n) \vdash x : \mathsf{Fin}(n)$$

- Separate *intuitionistic* / *unrestricted* uses and *linear* uses
- Types can depend on intuitionistic data, but not linear data

$$n : \mathsf{Nat} \mid x : \mathsf{Fin}(n) \vdash x : \mathsf{Fin}(n)$$

- ▷ Separate *intuitionistic* / *unrestricted* uses and *linear* uses
- ▷ Types can depend on intuitionistic data, but not linear data

(Barber, 1996)
(Cervesato and Pfenning, 2002)
(Krishnaswami, Pradic, and Benton, 2015)
(Vákár, 2015)

Quantitative Coeffect calculi:

$$x_1 \overset{\rho_1}{:} S_1, \ldots, x_n \overset{\rho_n}{:} S_n \vdash M : T$$

Quantitative Coeffect calculi:

$$x_1 \overset{\rho_1}{:} S_1, \ldots, x_n \overset{\rho_n}{:} S_n \vdash M : T$$

- The ρ_i record usage from some semiring R
  - $1 \in R$ — a use
  - $0 \in R$ — not used
  - $\rho_1 + \rho_2$ — adding up uses (e.g., in an application)
  - $\rho_1 \rho_2$ — nested uses

Quantitative Coeffect calculi:

$$x_1 \overset{\rho_1}{:} S_1, \ldots, x_n \overset{\rho_n}{:} S_n \vdash M : T$$

- The ρ_i record usage from some semiring R
  - $1 \in R$ — a use
  - $0 \in R$ — not used
  - $\rho_1 + \rho_2$ — adding up uses (e.g., in an application)
  - $\rho_1 \rho_2$ — nested uses

(Petricek, Orchard, and Mycroft, 2014)
(Brunel, Gaboardi, Mazza, and Zdancewic, 2014)
(Ghica and Smith, 2014)

Can we adapt this idea to dependent types?

Can we adapt this idea to dependent types?

McBride's idea:

- allow 0-usage data to appear in types.
  (McBride, 2016)

Can we adapt this idea to dependent types?

McBride's idea:

- allow 0-usage data to appear in types.
  (McBride, 2016)

$$x_1 \overset{\rho_1}{:} S_1, \ldots, x_n \overset{\rho_n}{:} S_n \vdash M \overset{\sigma}{:} T$$

where $\sigma \in \{0, 1\}$.

- $\sigma = 1$ — the "real" computational world
- $\sigma = 0$ — the types world

(allowing arbitrary ρ yields a system where substitution is inadmissible (Atkey, 2018))

Can we adapt this idea to dependent types?

McBride's idea:

- allow 0-usage data to appear in types.
  (McBride, 2016)

$$x_1 \overset{\rho_1}{:} S_1, \ldots, x_n \overset{\rho_n}{:} S_n \vdash M \overset{\sigma}{:} T$$

where $\sigma \in \{0, 1\}$.

- $\sigma = 1$ — the "real" computational world
- $\sigma = 0$ — the types world

(allowing arbitrary ρ yields a system where substitution is inadmissible (Atkey, 2018))

Zero-ing is an admissible rule: $\dfrac{\Gamma \vdash M \overset{1}{:} T}{0\Gamma \vdash M \overset{0}{:} T}$ allowing promotion to the type world.

Zero-ing is admissible

$$\frac{\Gamma \vdash M \overset{1}{:} T}{0\Gamma \vdash M \overset{0}{:} T}$$

means that every linear term has an “extensional” counterpart (or constitutent)

which can be used at type checking time to construct types

has the effect of making the linear system a restriction of the intuitionistic

A suitable semiring for affine linearity?

- Carrier: $\{0, 1, \omega\}$
- Ordered: $\omega < 1 < 0$
- Operations:

$+$	0	1	ω
0	0	1	ω
1	1	ω	ω
ω	ω	ω	ω

$\cdot$	0	1	ω
0	0	0	0
1	0	1	ω
ω	0	ω	ω

- Would admit an unrestricted $!$ modality.

Strict resource counting

- Carrier: $\mathbb{N}$
- Ordered: $\cdots < 2 < 1 < 0$
- Operations: normal operations on $\mathbb{N}$

Diamonds

$$\frac{\Gamma \vdash}{0\Gamma \vdash \Diamond}\ \textsc{Ty-Dia} \qquad \frac{0\Gamma \vdash}{0\Gamma \vdash * \overset{0}{:} \Diamond}\ \textsc{Tm-Dia}$$

- In the $\sigma = 0$ fragment, $\Diamond$s are free.

LFPL

- Natural number introduction

$$\frac{\Gamma \vdash d \overset{\sigma}{:} \diamond}{\Gamma \vdash \mathrm{zero}(d) \overset{\sigma}{:} \textsc{Nat}} \qquad \frac{\Gamma \vdash d \overset{\sigma}{:} \diamond \qquad \Gamma \vdash n \overset{\sigma}{:} \textsc{Nat}}{\Gamma \vdash \mathrm{succ}(d, n) \overset{\sigma}{:} \textsc{Nat}}$$

LFPL

- Natural number elimination ($\sigma = 1$ case)

$$\frac{\begin{array}{l} 0\Gamma, x : \textsc{Nat} \vdash A \\ \Gamma_1, d \overset{1}{:} \diamond \vdash M_z \overset{1}{:} A\{\mathrm{zero}(*)/x\} \\ d \overset{1}{:} \diamond, n \overset{0}{:} \textsc{Nat}, r \overset{1}{:} A\{n/x\} \vdash M_s \overset{1}{:} A\{\mathrm{succ}(*, n)/x\} \\ \Gamma_2 \vdash N \overset{1}{:} \textsc{Nat} \\ \Gamma_1 + \Gamma_2 = \Gamma \end{array}}{\Gamma \vdash \mathrm{iter}(x.A, d.M_z, d\ n\ r.M_s, N) \overset{1}{:} A\{N/x\}}$$

- Crucial: n is not available for computational use in M_s.

Encoding lists

- Define (in $\sigma = 0$ fragment):

$$\mathrm{Vec}\ A : \textsc{Nat} \longrightarrow \mathrm{Set}$$

  by iteration on the natural number.

- Lists:

$$\mathrm{List}\ A = (n \overset{1}{:} \textsc{Nat}) \otimes \mathrm{Vec}\ A\ n$$

Amortised Analysis

- Unrestricted introduction rules for natural numbers:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{zero} \overset{\sigma}{:} \textsc{Nat}} \qquad \frac{\Gamma \vdash N \overset{\sigma}{:} \textsc{Nat}}{\Gamma \vdash \mathsf{suc}(N) \overset{\sigma}{:} \textsc{Nat}}$$

- Postulate:

$$\Diamond^{(p_1,\ldots,p_k)} : \textsc{Nat} \to \mathrm{Set} \qquad \frac{\Gamma \vdash n \overset{0}{:} \textsc{Nat}}{\Gamma \vdash * \overset{0}{:} \Diamond^{(p_1,\ldots,p_k)}(n)}$$

- with:

$$\begin{aligned}
\mathrm{split} &: (n \overset{0}{:} \textsc{Nat}) \to \Diamond^{(p_1+p'_1,\ldots,p_k+p'_k)}(n) \multimap \Diamond^{(p_1,\ldots,p_k)}(n) \otimes \Diamond^{(p'_1,\ldots,p'_k)}(n) \\
\mathrm{join} &: (n \overset{0}{:} \textsc{Nat}) \to \Diamond^{(p_1,\ldots,p_k)}(n) \otimes \Diamond^{(p'_1,\ldots,p'_k)}(n) \multimap \Diamond^{(p_1+p'_1,\ldots,p_k+p'_k)}(n) \\
\mathrm{shift} &: (n \overset{0}{:} \textsc{Nat}) \to \Diamond^{(p_1,\ldots,p_k)}(\mathsf{suc}(n)) \multimap \Diamond^{(p_1+p_2,\ldots,p_k)}(n)
\end{aligned}$$

Amortised Analysis

- Natural number elimination ($\sigma = 1$ case)

$$\frac{\begin{array}{l} 0\Gamma, x \overset{0}{:} \textsc{Nat} \vdash A \\ \Gamma_1 \vdash M_z \overset{1}{:} A\{\mathrm{zero}/x\} \\ n \overset{1}{:} \textsc{Nat}, r \overset{1}{:} A\{n/x\} \vdash M_s \overset{1}{:} A\{\mathrm{succ}(n)/x\} \\ \Gamma_2 \vdash N \overset{1}{:} \textsc{Nat} \\ \Gamma_3 \vdash D \overset{1}{:} \diamond^{(1)}(N) \\ \Gamma_1 + \Gamma_2 + \Gamma_3 = \Gamma \end{array}}{\Gamma \vdash \mathrm{iter}(x.A, M_z, n\ r.M_s, N, D) : A\{N/x\}}$$

- n is available for use in M_s
- Pay up front for the iteration with D
- Get nested iteration by passing in enough $\diamond$s to pay for it

$$A[n] = \diamond^{(p_1, \ldots, p_k)}(n) \multimap B[n]$$

Semantic Interpretation : Soundness

Realisability for ICC (Dal Lago & Hofmann, 2011)

Resource monoids

- Let $\mathbb{N}_{-\infty}$ be category with objects $\mathbb{N} \cup \{-\infty\}$ and $m \longrightarrow n$ if $m \leq n$, with $-\infty \leq n$
  - Strict symmetric monoidal category with $(+, 0)$
- A resource monoid M is a $\mathbb{N}_{-\infty}$-enriched strict symmetric monoidal category.

Realisability for ICC (Dal Lago & Hofmann, 2011)

Resource monoids

- Let $\mathbb{N}_{-\infty}$ be category with objects $\mathbb{N} \cup \{-\infty\}$ and $m \longrightarrow n$ if $m \leq n$, with $-\infty \leq n$
    - Strict symmetric monoidal category with $(+, 0)$
- A resource monoid M is a $\mathbb{N}_{-\infty}$-enriched strict symmetric monoidal category.

- $(M, +, 0)$ is a commutative monoid
- $0 \leq M(\alpha, \alpha)$
- $M(\alpha, \beta) \in \mathbb{N}_{-\infty}$ is the difference between α and β
- $M(\alpha, \beta) + M(\beta, \gamma) \leq M(\alpha, \gamma)$
- $M(\alpha, \beta) \leq M(\alpha + \gamma, \beta + \gamma)$

Resource monoids

Linear time:

- $M = \mathbb{N}$
- Differencing:

$$M(n, m) = \begin{cases} m - n & n \leq m \\ -\infty & \text{otherwise} \end{cases}$$

- Wrinkle: counts recursion steps, not the actual number of steps.

Resource Monoids: Polynomial time (for LFPL)

- $M \ni (n, p)$, where
  - $n \in \mathbb{N}$ is the amount of iterability (number of $\diamond$s)
  - p is a polynomial with $\mathbb{N}$ coefficients
  - $(n, p) + (m, q) = (n + m, p + q)$.
  - Cost differencing:

$$M((n,p),(m,q)) = \begin{cases} q(m) - p(m) & n \leq m \text{ and } (q - p) \text{ is non-negative} \\ & \quad \text{and non-decreasing } \geq m \\ -\infty & \text{otherwise} \end{cases}$$

Resource Monoids: Polynomial time (for Constructor-free System)

- $M \ni (n, p)$, where
  - $n \in \mathbb{N}$ is the amount of iterability (number of $\diamond$s)
  - p is a polynomial with $\mathbb{N}$ coefficients
  - $(n, p) + (m, q) = (\max n\ m, p + q)$.
  - Cost differencing:

$$M((n, p), (m, q)) = \begin{cases} q(m) - p(m) & n \leq m \text{ and } (q - p) \text{ is non-negative} \\ & \quad \text{and non-decreasing } \geq m \\ -\infty & \text{otherwise} \end{cases}$$

- Hofmann and Dal Lago used this resource monoid for Lafont's *Soft Linear Logic*.

Cost model

- Assume a model of computation with a cost model:

$$e, \eta \Downarrow_k v$$

step count k, expressions $e \in \mathcal{E}$, values $v \in \mathcal{V}$.

Interpretation of Types and Terms

- Types are interpreted by $(|X|, \models_X)$ where:
  - $|X|$ is a set
  - $\models_X \subseteq (M \times \mathcal{V}) \times |X|$.

- Functions $f : X \longrightarrow Y$:
  - $f : |X| \longrightarrow |Y|$
  - exists $e \in \mathcal{E}$, $\gamma \in M$, such that
  - for all α, v, x.

    $(\alpha, v) \models_X x$ implies

      exists β, k, v' s.t.

        $e, [v] \Downarrow_k v'$,

        $(\beta, v') \models_Y f(x)$,

        $k \leq M(\alpha + \gamma, \beta)$

Some types

In the amortised system:

- $\diamond = (\{*\}, (n, *) \models_\diamond * \Leftrightarrow n \geq 1)$

In LFPL:

- $\diamond = (\{*\}, ((n, p), * \models_\diamond * \Leftrightarrow n \geq 1, p \geq 0)$
- $\textsc{Nat} = (\mathbb{N}, ((n, p), n \models_\diamond \underline{m}) \Leftrightarrow n \geq m, p \geq 0)$

In the constructor free system:

- $\textsc{Nat} = (\mathbb{N}, ((n, p), n \models_\diamond \underline{m}) \Leftrightarrow n \geq m, p \geq 0)$

Summary

- Quantitative Type Theory for Complexity Analysis
- Careful combination of dependency and linearity
- Dependent Types for reasoning about programs
- Dependent Types for reasoning about complexity (in the explicit system)

- ▷ Quantitative Type Theory for Complexity Analysis
- ▷ Careful combination of dependency and linearity
- ▷ Dependent Types for reasoning about programs
- ▷ Dependent Types for reasoning about complexity (in the explicit system)

Related Work

- Sized types
  Used for controlling well foundedness
  For complexity analysis require “tick” monads
- Gaboardi and Dal Lago: Linear Dependent Types for ICC
  Dependent Types only for counting time
- Future:
  - LAL, EAL, BLL, Logspace, ...
  - Polytime mathematics?