Towards a logical foundation of computational complexity

Kazushige Terui

RIMS, Kyoto University

email: terui@kurims.kyoto-u.ac.jp

Background

- Logic and type theory. Proofs = Programs.
- Complexity issues arise in
  - type checking/inference,
  - verification,
  - normalization, etc.
- Implicit computational complexity:

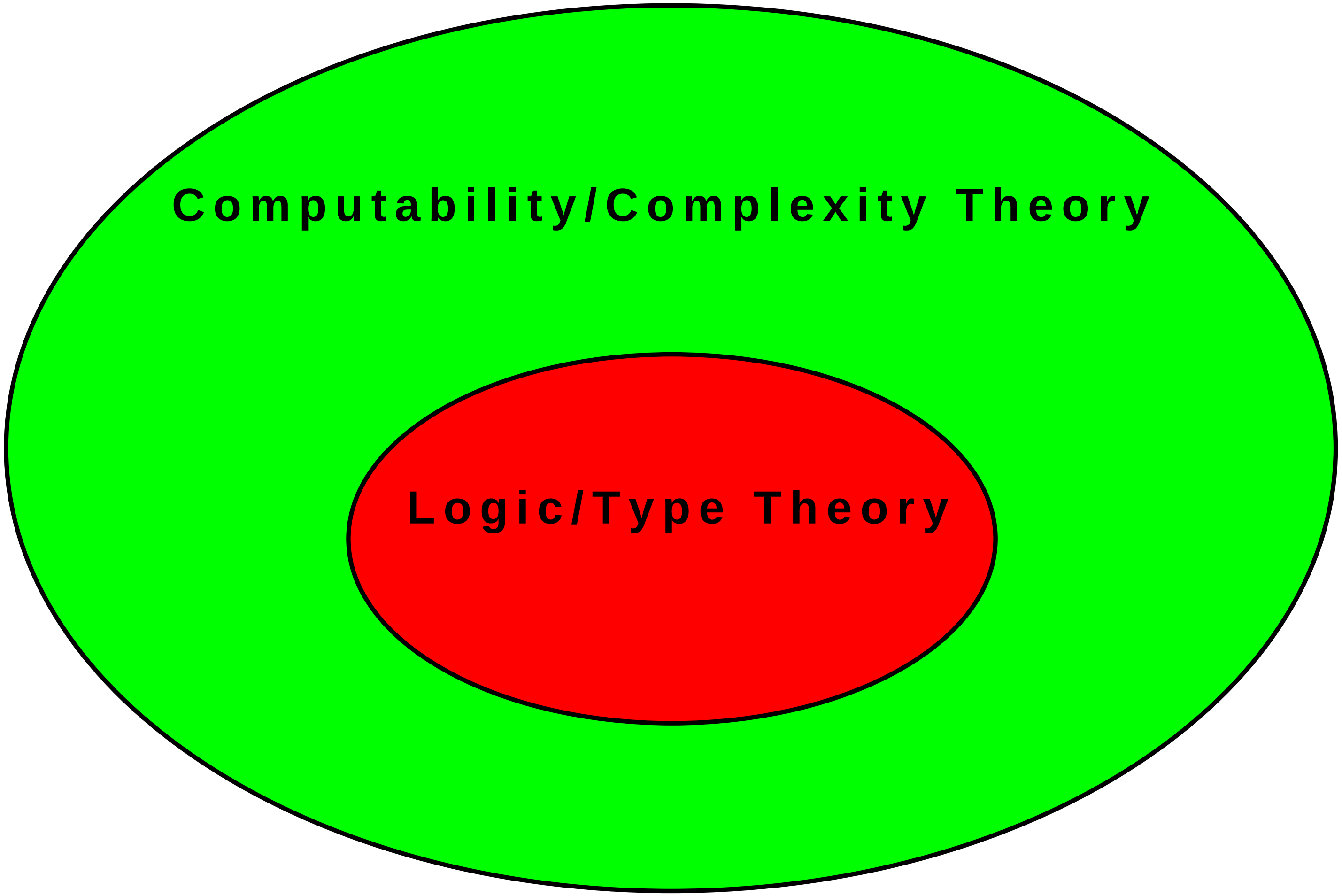
  Machine-independent, parameter-free characterizations of complexity classes (such as P)

Background

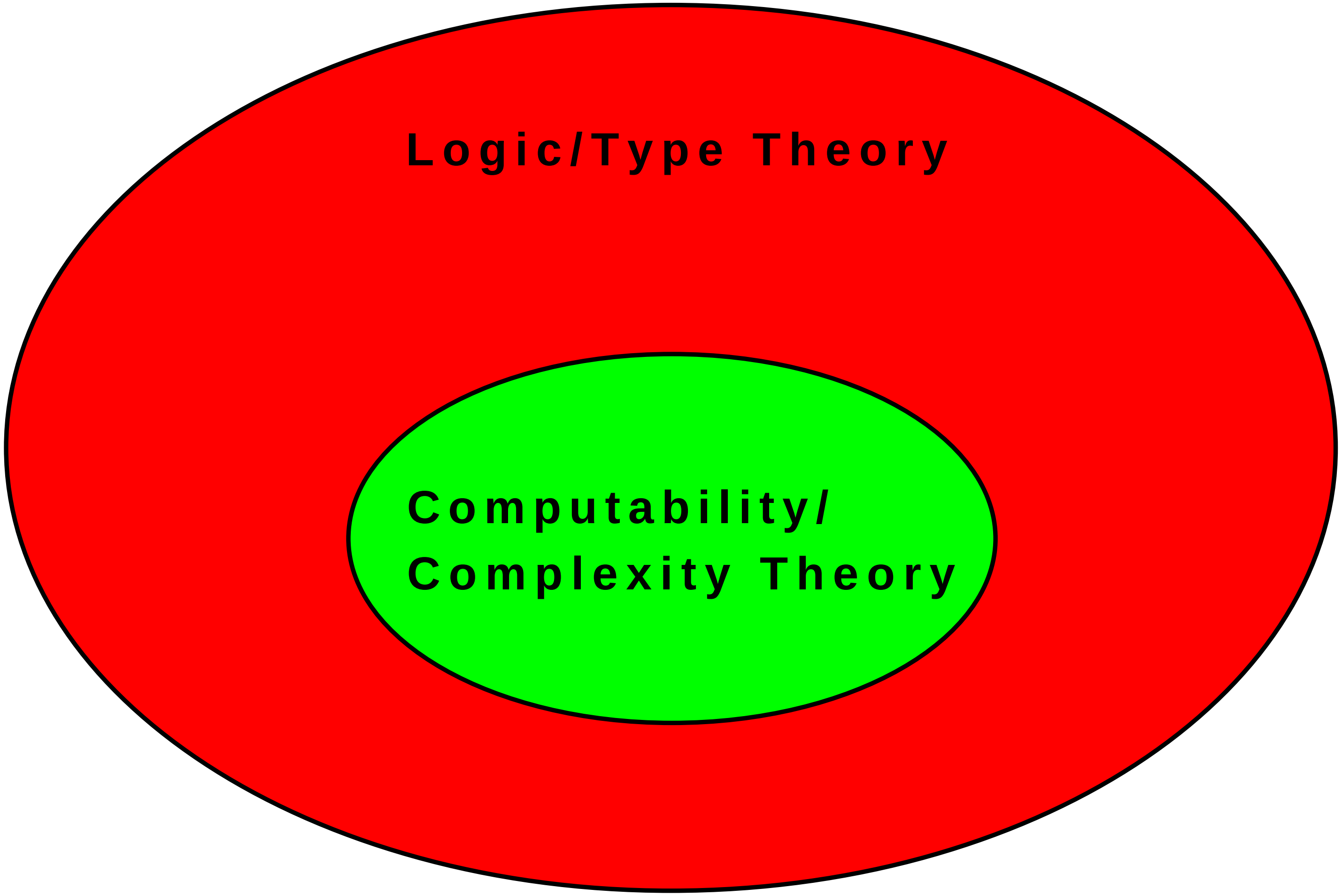
- Gödel's T + restrictions:
  - Safety (Bellantoni-Cook, Leivant, Marion, . . .)
  - Linearity at higher order (Bellantoni-Niggl-Schwichtenberg, Hofmann, . . .)
  - Cons-free (Jones, Kristiansen, . . .)
- Girard's F + restrictions:
  - Light linear logic (Girard, Asperti, Baillot-T., . . .)
  - Soft linear logic (Lafont, Gaboardi-Ronchi, Hofmann-Schöpp, . . .)
- Complexity of simply typed lambda calculus (Schubert)
- Complexity of fragments of linear logic (Mairson-T.)
- Parallel complexity of proof nets (T.)

Current Status

When talking about complexity of proofs and programs ...



Our Ambition



Logic/Type Theory

Computability/
Complexity Theory

Our project

- Explain basic phenomena in computability and complexity from the view point of logic and type theory.

Our project

- Explain basic phenomena in computability and complexity from the view point of logic and type theory.
  1. Reconstruct basic objects of C & C (machines, languages) as logical objects (proofs, types)

Our project

- Explain basic phenomena in computability and complexity from the view point of logic and type theory.
  1. Reconstruct basic objects of C & C (machines, languages) as logical objects (proofs, types)
  2. Derive C & C theorems as corollaries of (meta)theorems of logic

Our project

- Explain basic phenomena in computability and complexity from the view point of logic and type theory.

1. Reconstruct basic objects of C & C (machines, languages) as logical objects (proofs, types)
2. Derive C & C theorems as corollaries of (meta)theorems of logic
3. Take full advantage of generality (various data/higher order) and type-based reasoning (type isomorphisms/logic metatheorems)

Our project

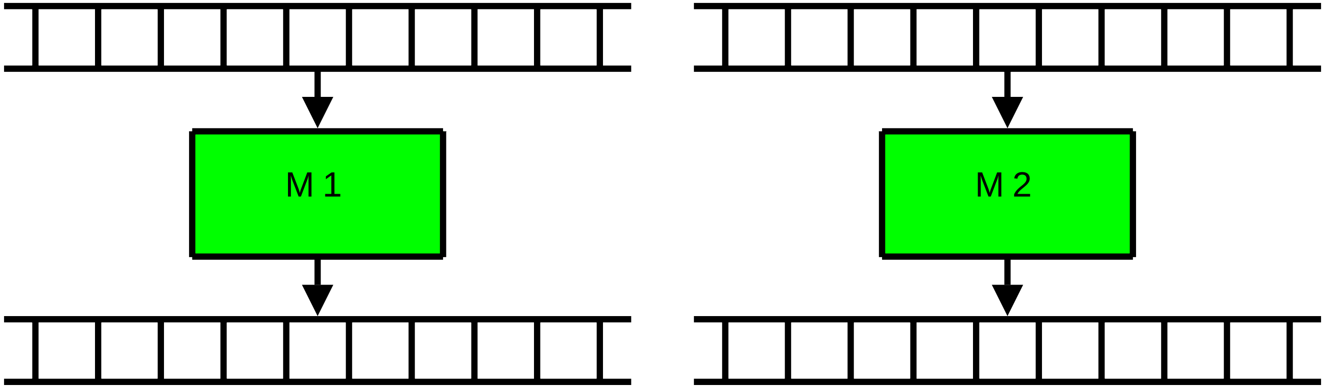
- Explain basic phenomena in computability and complexity from the view point of logic and type theory.
  1. Reconstruct basic objects of C & C (machines, languages) as logical objects (proofs, types)
  2. Derive C & C theorems as corollaries of (meta)theorems of logic
  3. Take full advantage of generality (various data/higher order) and type-based reasoning (type isomorphisms/logic metatheorems)
- Which logic? — Ludics (Girard 2001).

Outline

1. Time and space sensitive compositions in lambda calculus
2. What is ludics?
3. Data and computation in ludics
4. Arbitrary data sets
5. Language operators and internal completeness
6. Space compression and focalization
7. Conclusion

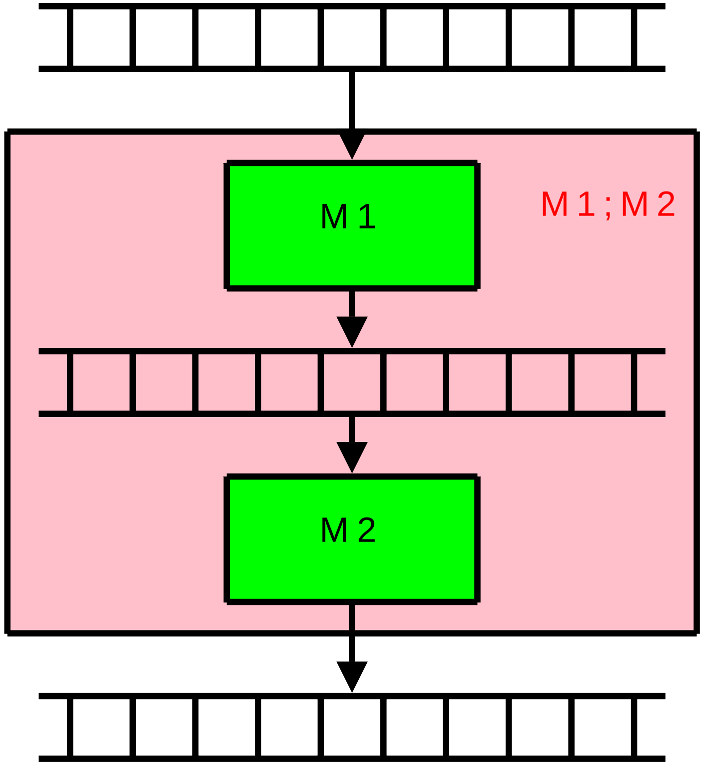
Composition of TMs

How do you compose two TMs?



Composition of TMs

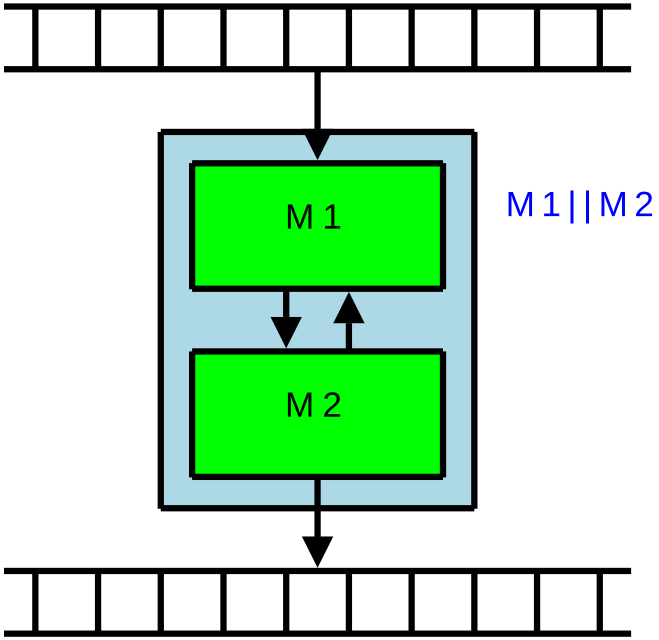
- Sequential composition $M_1; M_2$: first simulate M_1, then M_2



- Time efficient, but not space efficient.

Composition of TMs

- Interactive composition $M_1||M_2$: simulate a dialogue between M_1 and M_2



- Space efficient, but not time efficient.

Evaluations of lambda terms

- Lambda calculus admits a canonical composition:

$$t \circ u = \lambda x.t(u(x)).$$

Evaluations of lambda terms

- Lambda calculus admits a canonical composition:

$$t \circ u = \lambda x.t(u(x)).$$

- There are various evaluation methods.

Evaluations of lambda terms

- Lambda calculus admits a canonical composition:

$$t \circ u = \lambda x.t(u(x)).$$

- There are various evaluation methods.
- Call-by-value β-reduction

Evaluations of lambda terms

- Lambda calculus admits a canonical composition:

$$t \circ u = \lambda x.t(u(x)).$$

- There are various evaluation methods.
- Call-by-value β-reduction
  - Simulates TMs with a quadratic time/linear space overhead (cf. Dal Lago-Martini).

Evaluations of lambda terms

- Lambda calculus admits a canonical composition:

$$t \circ u = \lambda x.t(u(x)).$$

- There are various evaluation methods.
- Call-by-value β-reduction
  - Simulates TMs with a quadratic time/linear space overhead (cf. Dal Lago-Martini).
  - Composition is time efficient, but not space efficient.

Evaluations of lambda terms

- Lambda calculus admits a canonical composition:

$$t \circ u = \lambda x.t(u(x)).$$

- There are various evaluation methods.
- Call-by-value β-reduction
  - Simulates TMs with a quadratic time/linear space overhead (cf. Dal Lago-Martini).
  - Composition is time efficient, but not space efficient.
- Is there a space efficient evaluation method?

Krivine's Abstract Machine

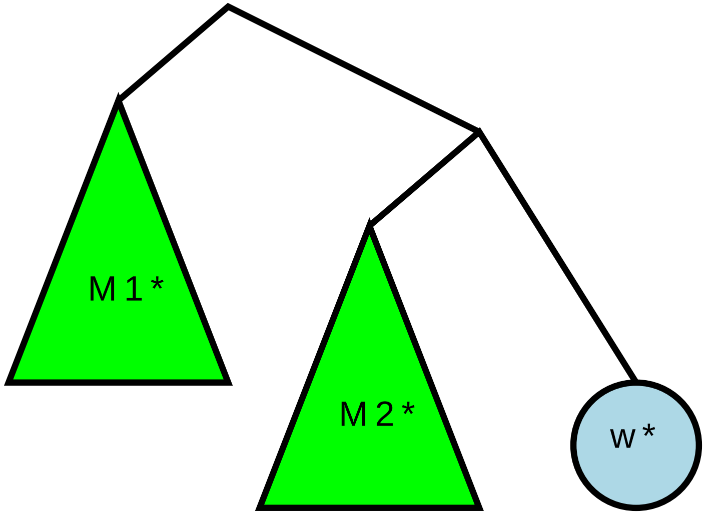
- A pointer machine working on (graphs of) untyped λ-terms
- Equipped with environments ρ (for variables) and stacks π (of arguments)

$$\begin{array}{rcll} (x\rho,\ \pi) & \longrightarrow & (\rho(x),\ \pi) & \text{if } x \in Dom(\rho) \\ ((tu)\rho,\ \pi) & \longrightarrow & (t\rho,\ u\rho : \pi) & \\ ((\lambda x.t)\rho,\ u\rho' : \pi) & \longrightarrow & (t\rho[x \mapsto u\rho'],\ \pi) & \end{array}$$

- Fact: There is no evaluator that is significantly and uniformly more space-efficient than (optimized) KAM.

Time-space tradeoff in λ-calculus

- Compose encodings M_1^*, M_2^* of TMs M_1 and M_2:



- CBV simulates sequential composition $M_1; M_2$
- KAM simulates interactive composition $M_1 || M_2$
- Time-space tradeoff shows up in a different way in lambda calculus.

Outline

1. Time and space sensitive compositions in lambda calculus
2. What is ludics?
3. Data and computation in ludics
4. Arbitrary data sets
5. Language operators and internal completeness
6. Space compression and focalization
7. Conclusion

What is ludics?

- Ludics (Girard 01): pre-logical framework upon which logic is built and various phenomena are analyzed.

What is ludics?

- Ludics (Girard 01): pre-logical framework upon which logic is built and various phenomena are analyzed.
- Keywords: Monism, existentialism, interaction/orthogonality:

What is ludics?

- Ludics (Girard 01): pre-logical framework upon which logic is built and various phenomena are analyzed.
- Keywords: Monism, existentialism, interaction/orthogonality:

Game Semantics	⟺	Ludics	⟺	Proof Theory
strategies		designs		proofs
⇑		⇓orthogonality		⇑
games		behaviours		types

What is ludics?

- Ludics (Girard 01): pre-logical framework upon which logic is built and various phenomena are analyzed.
- Keywords: Monism, existentialism, interaction/orthogonality:

Game Semantics	$\Longleftrightarrow$	Ludics	$\Longleftrightarrow$	Proof Theory
strategies		designs		proofs
$\Uparrow$		$\Downarrow$orthogonality		$\Uparrow$
games		behaviours		types

- Orthogonality $P \perp N$: “Players P and N well socialize”

What is ludics?

- Ludics (Girard 01): pre-logical framework upon which logic is built and various phenomena are analyzed.
- Keywords: Monism, existentialism, interaction/orthogonality:

Game Semantics	$\Longleftrightarrow$	Ludics	$\Longleftrightarrow$	Proof Theory
strategies		designs		proofs
$\Uparrow$		$\Downarrow$orthogonality		$\Uparrow$
games		behaviours		types

- Orthogonality $P \perp N$: “Players P and N well socialize”
- Construction of behaviours: $\{P\}^{\perp}$, $\mathbf{P}^{\perp} = \mathbf{N}$, $\mathbf{N}^{\perp} = \mathbf{P}$
  “Pair $(\mathbf{P}, \mathbf{N})$ of two player sets form a game.”

From C & C to Ludics

- Basic concepts and constructions in C & C:

From C & C to Ludics

- Basic concepts and constructions in C & C:
  - Machine M accepts $w \in \Sigma^*$

From C & C to Ludics

- Basic concepts and constructions in C & C:
  - Machine M accepts $w \in \Sigma^*$
  - $L(M) = \{w : M \text{ accepts } w\}$

From C & C to Ludics

- Basic concepts and constructions in C & C:
  - Machine M accepts $w \in \Sigma^*$
  - $L(M) = \{w : M \text{ accepts } w\}$
  - $L_1 \cup L_2 = L(M_0)$

From C & C to Ludics

- Basic concepts and constructions in C & C:
  - Machine M accepts $w \in \Sigma^*$
  - $L(M) = \{w : M \text{ accepts } w\}$
  - $L_1 \cup L_2 = L(M_0)$
- In ludics:

From C & C to Ludics

- Basic concepts and constructions in C & C:
  - Machine M accepts $w \in \Sigma^*$
  - $L(M) = \{w : M \text{ accepts } w\}$
  - $L_1 \cup L_2 = L(M_0)$
- In ludics:
  - $M^\bullet \perp w^\bullet$ (orthogonality)

From C & C to Ludics

- Basic concepts and constructions in C & C:
  - Machine M accepts $w \in \Sigma^*$
  - $L(M) = \{w : M \text{ accepts } w\}$
  - $L_1 \cup L_2 = L(M_0)$
- In ludics:
  - $M^\bullet \perp w^\bullet$ (orthogonality)
  - $\{M^\bullet\}^\perp \doteq L(M)$ (behaviour)

From C & C to Ludics

- Basic concepts and constructions in C & C:
  - Machine M accepts $w \in \Sigma^*$
  - $L(M) = \{w : M \text{ accepts } w\}$
  - $L_1 \cup L_2 = L(M_0)$
- In ludics:
  - $M^\bullet \perp w^\bullet$ (orthogonality)
  - $\{M^\bullet\}^\perp \doteq L(M)$ (behaviour)
  - $a.L_1 \cup b.L_2 \doteq (a.L_1 \cup b.L_2)^{\perp\perp}$ (internal completeness)

From C & C to Ludics

- Basic concepts and constructions in C & C:
  - Machine M accepts $w \in \Sigma^*$
  - $L(M) = \{w : M \text{ accepts } w\}$
  - $L_1 \cup L_2 = L(M_0)$
- In ludics:
  - $M^\bullet \perp w^\bullet$ (orthogonality)
  - $\{M^\bullet\}^\perp \doteq L(M)$ (behaviour)
  - $a.L_1 \cup b.L_2 \doteq (a.L_1 \cup b.L_2)^{\perp\perp}$ (internal completeness)
- There is no ontological distinction between $M^\bullet$ and $w^\bullet$. $\perp$ is homogeneous and symmetric.

Computational Ludics

- We introduce a modified version: computational ludics.
  - Absolute addresses $\Longrightarrow$ Term calculus with variable binding
  - No care of finiteness $\Longrightarrow$ Sensitive to finite generation
  - Cut-free designs $\Longrightarrow$ Cut-ful ones

Well-behaved frag. of simply typed λ-calculus

- Types: $\tau ::= \iota \mid \tau \to \tau$
- Positive terms P and negative terms N are defined by:

$$\begin{array}{rcl} P^{\iota} & ::= & (N_0^{\tau_1 \to \dots \tau_n \to \iota}) N_1^{\tau_1} \dots N_n^{\tau_n} \\ N^{\tau_1 \to \dots \tau_n \to \iota} & ::= & x \mid \lambda x_1^{\tau_1} \cdots x_n^{\tau_n} . P^{\iota} \end{array}$$

- Reduction: the arity n always agrees.

$$(\lambda x_1 \cdots x_n . P) N_1 \cdots N_n \;\longrightarrow\; P[N_1/x_1, \dots, N_n/x_n]$$

Towards ludics

- Designs in ludics:
  - Type-free; arity agreement is ensured in another way.
  - Infinitary (coinduitive).
  - Daimon (immediate termination)
  - Additive superimposition: $N_1 + N_2 + N_3 + \cdots$
  - Various actions (rather than the single pair λ/@) given by a signature.
- Signature: $\mathcal{A} = (A, ar)$

  A is a set of names,

  $ar : A \longrightarrow \mathcal{N}$ gives an arity to each name.

Computational designs

- The set of designs is coinductively defined by:

$$\begin{array}{rcll} P & ::= & \maltese & \text{Daimon} \\ & \mid & \Omega & \text{Divergence} \\ & \mid & N_0|\overline{a}\langle N_1, \ldots, N_n\rangle & \text{Proper positive action} \\ N & ::= & x & \text{Variable} \\ & \mid & \sum a(\vec{x}_a).P_a & \text{Proper negative action} \end{array}$$

  - where $ar(a) = n$, $\vec{x}_a = x_1, \ldots, x_n$
  - $\sum a(\vec{x}_a).P_a$ is built from $\{a(\vec{x}_a).P_a\}_{a \in A}$. Compare it with:

$$\begin{array}{rcl} P & ::= & (N_0)N_1 \ldots N_n \\ N & ::= & x \mid \lambda x_1 \cdots x_n.P \end{array}$$

Normalization 1

- Designs $\doteq$ Processes in linear π-calculus (Faggian-Piccolo)

Normalization 1

- Designs $\doteq$ Processes in linear π-calculus (Faggian-Piccolo)
- Ω allows partial branching:

$$a(\vec{x}).P + b(\vec{y}).Q \;=\; a(\vec{x}).P + b(\vec{y}).Q + c(\vec{z}).\Omega + d(\vec{z}).\Omega + \cdots$$

Normalization 1

- Designs $\doteq$ Processes in linear π-calculus (Faggian-Piccolo)
- Ω allows partial branching:

$$a(\vec{x}).P + b(\vec{y}).Q \;=\; a(\vec{x}).P + b(\vec{y}).Q + c(\vec{z}).\Omega + d(\vec{z}).\Omega + \cdots$$

- Reduction rule:

$$(\sum a(x_1, \ldots, x_n).P_a) \,|\, \overline{a}\langle N_1, \ldots, N_n\rangle \;\longrightarrow\; P_a[N_1/x_1, \ldots, N_n/x_n].$$

Normalization 1

- Designs $\doteq$ Processes in linear π-calculus (Faggian-Piccolo)
- Ω allows partial branching:

$$a(\vec{x}).P + b(\vec{y}).Q \;=\; a(\vec{x}).P + b(\vec{y}).Q + c(\vec{z}).\Omega + d(\vec{z}).\Omega + \cdots$$

- Reduction rule:

$$(\sum a(x_1, \ldots, x_n).P_a) \,|\overline{a}\langle N_1, \ldots, N_n\rangle \;\longrightarrow\; P_a[N_1/x_1, \ldots, N_n/x_n].$$

- Compare it with

$$(\lambda x_1 \cdots x_n.P)N_1 \cdots N_n \;\longrightarrow\; P[N_1/x_1, \ldots, N_n/x_n]$$

Orthogonality

- A positive design P is one of the following forms:

$x\|\overline{a}\langle N_1,\ldots,N_n\rangle$	Head normal form
$(\sum a(\vec{x}_a).P_a)\,\|\overline{a}\langle N_1,\ldots,N_n\rangle$	Cut
$\maltese$	Daimon
Ω	Divergence

- Fact: For any closed positive design P,

$$P \longrightarrow^* \maltese \text{ or diverges.}$$

- Orthogonality: Suppose $fv(P) \subseteq \{x_0\}$ and $fv(N) = \emptyset$.

$$P \perp N \quad\Longleftrightarrow\quad P[N/x_0] \Downarrow \maltese.$$

Normalization: the general case

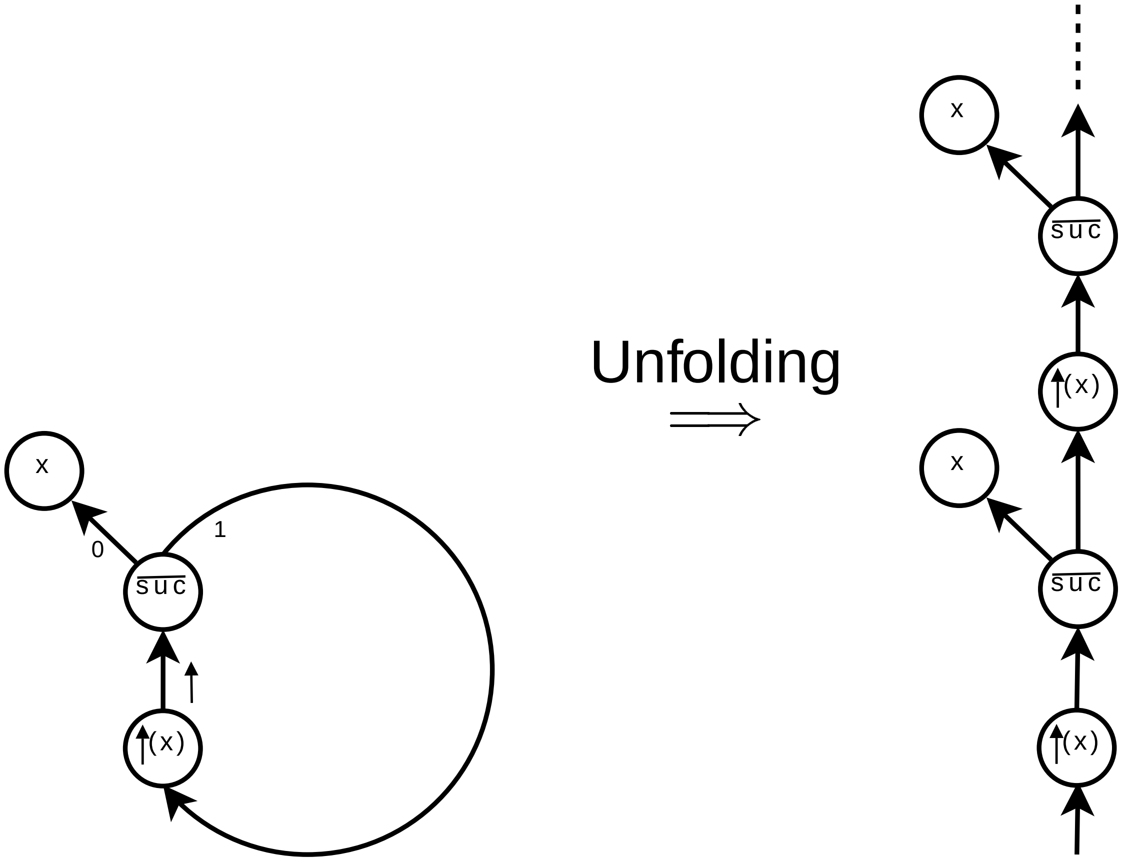
- Head reduction: $(\sum a(\vec{x}_a).P_a)\,|\overline{a}\langle\vec{N}_a\rangle \longrightarrow P_a[\vec{N}_a/\vec{x}]$.
- By corecursion, it can be extended to $[\![\ ]\!]$:

$$
\begin{array}{rcll}
[\![P]\!] & = & \maltese & \text{if } P \Downarrow \maltese; \\
 & = & x|\overline{a}\langle [\![N_1]\!],\ldots,[\![N_n]\!]\rangle & \text{if } P \Downarrow x|\overline{a}\langle N_1,\ldots,N_n\rangle; \\
 & = & \Omega & \text{if } P \Uparrow; \\
[\![x]\!] & = & x; & \\
[\![\sum a(\vec{x}_a).P_a]\!] & = & \sum a(\vec{x}_a).[\![P_a]\!]. &
\end{array}
$$

- Non-effective: it works on infinite designs; renaming and substitution involved.

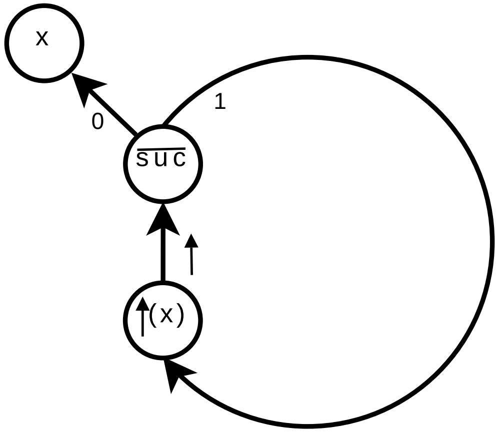
Finite generation

- Finite generation: Some infinite l-designs can be obtained from a finite graph by unfolding:



Finite generation

- Is there any normalization procedure that directly works on graph representations?



- Krivine’s abstract machine can be adapted to do so.

L-designs

- P is total if $P \neq \Omega$.
- T is linear if for any subterm $N_0|a\langle N_1, \ldots, N_n\rangle$, $fv(N_0), \ldots, fv(N_n)$ are pairwise disjoint.
- x is identity if it occurs in a bracket $N_0|\overline{a}\langle N_1, \ldots, x, \ldots, N_n\rangle$.
- L-designs: total, linear, identity-free designs with finitely many free variables.

Outline

1. Time and space sensitive compositions in lambda calculus
2. What is ludics?
3. Data and computation in ludics
4. Arbitrary data sets
5. Language operators and internal completeness
6. Space compression and focalization
7. Conclusion

What are data?

- Examples: integers, words, trees, lists, records, etc.
- Data must be:
  - structured (eg. list = head + tail)
  - linearly duplicable (“linear” = “machine-like”)
  - compressable (eg. binary int. $\rightarrow$ hexadecimal int.)
- Fix a unary name $\uparrow \in A$.
- Notation: $\downarrow = \overline{\uparrow}$, $\qquad \uparrow\overline{a}\langle\vec{N}\rangle = \uparrow(x).x|\overline{a}\langle\vec{N}\rangle$
- The set of data designs is coinductively defined by

$$d \ ::= \uparrow\overline{a}\langle d, \ldots, d\rangle, \qquad a \in A.$$

Data: examples

- Natural numbers

$$\begin{aligned} 0^\bullet &= {\uparrow}\overline{\mathsf{zero}} \\ n+1^\bullet &= {\uparrow}\overline{\mathsf{suc}}\langle n^\bullet \rangle \end{aligned}$$

- Ordinal omega

$$\omega^\bullet = {\uparrow}\overline{\mathsf{suc}}\langle \omega^\bullet \rangle.$$

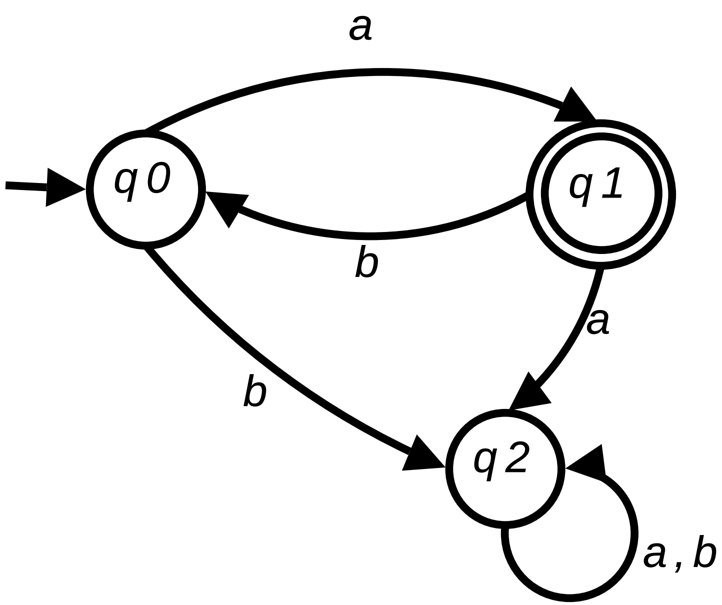
- Words, labelled binary trees, and lists:

$$\begin{aligned} \epsilon^\bullet &= {\uparrow}\overline{\mathsf{nil}} \\ aba^\bullet &= {\uparrow}\overline{a}\langle {\uparrow}\overline{b}\langle {\uparrow}\overline{a}\langle {\uparrow}\overline{\mathsf{nil}}\rangle\rangle\rangle \\ \mathtt{node}_a(\mathtt{leaf}_b, \mathtt{leaf}_c)^\bullet &= {\uparrow}\overline{a}\langle {\uparrow}\overline{b}, {\uparrow}\overline{c}\rangle \end{aligned}$$

- Infinite words and trees are also representable.

From DFAs to cut-free l-designs

A DFA accepting $a(ba)^*$:



$$\begin{array}{rclcrcl}
P_0 & = & x|{\downarrow}\langle N_0\rangle, & \qquad & N_0 & = & a(x).P_1 + b(x).P_2 + \mathsf{nil}.\Omega, \\
P_1 & = & x|{\downarrow}\langle N_1\rangle, & \qquad & N_1 & = & a(x).P_2 + b(x).P_0 + \mathsf{nil}.\maltese, \\
P_2 & = & x|{\downarrow}\langle N_2\rangle, & \qquad & N_2 & = & a(x).P_2 + b(x).P_2 + \mathsf{nil}.\Omega.
\end{array}$$

From DFAs to cut-free l-designs

$$\begin{array}{rcl@{\qquad\qquad}rcl}
P_0 & = & x|{\downarrow}\langle N_0\rangle, & N_0 & = & a(x).P_1 + b(x).P_2 + \mathsf{nil}.\Omega, \\
P_1 & = & x|{\downarrow}\langle N_1\rangle, & N_1 & = & a(x).P_2 + b(x).P_0 + \mathsf{nil}.\maltese, \\
P_2 & = & x|{\downarrow}\langle N_2\rangle, & N_2 & = & a(x).P_2 + b(x).P_2 + \mathsf{nil}.\Omega.
\end{array}$$

$$\begin{array}{rcl}
P_0[aba^\bullet/x] & = & P_0[\,{\uparrow}(x).x|\overline{a}\langle ba^\bullet\rangle \;/\; x\,] \\
& \longrightarrow & N_0|\overline{a}\langle ba^\bullet\rangle \\
& \longrightarrow & P_1[\; ba^\bullet \;/\; x\;] \\
& \longrightarrow^* & P_0[\; a^\bullet \;/\; x\;] \\
& \longrightarrow^* & P_1[\,{\uparrow}(x).x|\overline{\mathsf{nil}} \;/\; x\,] \\
& \longrightarrow & N_1|\overline{\mathsf{nil}} \\
& \longrightarrow^* & \maltese
\end{array}$$

DFAs = f.g. cut-free l-designs

- Theorem: DFA M $\rightleftharpoons$ finitely generated cut-free l-design P:

$$\text{M accepts } w \iff P \perp w^{\bullet},$$

for every $w \in \Sigma^*$.

DFAs = f.g. cut-free l-designs

- Theorem: DFA M $\rightleftharpoons$ finitely generated cut-free l-design P:

  $$\text{M accepts } w \iff P \perp w^{\bullet},$$

  for every $w \in \Sigma^*$.

- Finitely generated cut-free l-designs capture the regular languages.

DFAs = f.g. cut-free l-designs

- Theorem: DFA M $\rightleftharpoons$ finitely generated cut-free l-design P:

  $$\text{M accepts } w \iff P \perp w^\bullet,$$

  for every $w \in \Sigma^*$.
- Finitely generated cut-free l-designs capture the regular languages.
- Too weak!

DFAs = f.g. cut-free l-designs

- Theorem: DFA M $\rightleftharpoons$ finitely generated cut-free l-design P:

  $$\text{M accepts } w \iff P \perp w^{\bullet},$$

  for every $w \in \Sigma^*$.
- Finitely generated cut-free l-designs capture the regular languages.
- Too weak!
- To enrich automata, one equips them with stacks.

DFAs = f.g. cut-free l-designs

- Theorem: DFA M $\rightleftharpoons$ finitely generated cut-free l-design P:

  $$\text{M accepts } w \iff P \perp w^{\bullet},$$

  for every $w \in \Sigma^*$.
- Finitely generated cut-free l-designs capture the regular languages.
- Too weak!
- To enrich automata, one equips them with stacks.
- To enrich designs, one equips them with cuts.

L-designs with cuts

- Succesors, Discriminators
- Duplicator $Dup[x]$. For any finite data design d,

$$[\![Dup[d]]\!] = \uparrow\overline{\mathsf{pair}}(d,d).$$

- Cf. Duplicators in linear λ-calculus (with limited rec.)

$$\begin{array}{llll}
Dup_{\mathsf{B}}(x) = & \mathit{case} & x = \mathsf{true} \Rightarrow & \mathsf{true} \otimes \mathsf{true} \\
 & & x = \mathsf{false} \Rightarrow & \mathsf{false} \otimes \mathsf{false} \\
Dup_{\mathsf{N}}(x) = & \mathit{case} & x = \mathsf{zero} \Rightarrow & \mathsf{zero} \otimes \mathsf{zero} \\
 & & x = \mathsf{suc}(y) \Rightarrow & \mathit{let}\ z_1 \otimes z_2 = Dup_{\mathsf{N}}(y) \\
 & & & \quad \mathit{in}\ \ \mathsf{suc}(z_1) \otimes \mathsf{suc}(z_2)
\end{array}$$

- Cut is essential for finite generation.
- Q: Does Dup duplicate $\omega^{\bullet}$?

L-designs with cuts

- Theorem: TM M $\rightleftharpoons$ finitely generated (cut-ful) l-design P:

$$\text{M accepts } w \iff P \perp w^{\bullet}.$$

- Proof.

(⇒) Successors, discriminators, duplicators and the general recursion scheme are available with cuts.

(⇐) Krivine's abstract machine works effectively on finite graph representations.

- Finitely generated cut-ful l-designs capture the r.e. languages.
- What about arbitrary data sets?

Outline

1. Time and space sensitive compositions in lambda calculus
2. What is ludics?
3. Data and computation in ludics
4. Arbitrary data sets
5. Language operators and internal completeness
6. Space compression and focalization
7. Conclusion

Strong separation for data designs

- Böhm's theorem in lambda calculus: Given $t \neq_{\beta\eta} u$, there is $C[]$ such that $C[t] =_{\beta\eta} \lambda xy.x$ and $C[u] =_{\beta\eta} \lambda xy.y$.

Strong separation for data designs

- Böhm's theorem in lambda calculus: Given $t \neq_{\beta\eta} u$, there is $C[]$ such that $C[t] =_{\beta\eta} \lambda xy.x$ and $C[u] =_{\beta\eta} \lambda xy.y$.
- Separation theorem in ludics: for any (atomic) cut-free N, M, $N = M$ iff $P \perp N \iff P \perp M$ for any P

Strong separation for data designs

- Böhm's theorem in lambda calculus: Given $t \neq_{\beta\eta} u$, there is $C[]$ such that $C[t] =_{\beta\eta} \lambda xy.x$ and $C[u] =_{\beta\eta} \lambda xy.y$.
- Separation theorem in ludics: for any (atomic) cut-free N, M, $N = M$ iff $P \perp N \iff P \perp M$ for any P
- Strong separation for finite data designs:
  for any finite data design d, there is a counter design d^c such that for any e,

$$d = e \quad \text{iff} \quad d^c \perp e.$$

Strong separation for data designs

- Böhm's theorem in lambda calculus: Given $t \neq_{\beta\eta} u$, there is $C[]$ such that $C[t] =_{\beta\eta} \lambda xy.x$ and $C[u] =_{\beta\eta} \lambda xy.y$.
- Separation theorem in ludics: for any (atomic) cut-free N, M, $N = M$ iff $P \perp N \iff P \perp M$ for any P
- Strong separation for finite data designs: for any finite data design d, there is a counter design d^c such that for any e,
$$d = e \quad \text{iff} \quad d^c \perp e.$$
- "For any $w \in \Sigma^*$ there is a DFA M such that $L(M) = \{w\}$."

Strong separation for data designs

- Böhm's theorem in lambda calculus: Given $t \neq_{\beta\eta} u$, there is $C[]$ such that $C[t] =_{\beta\eta} \lambda xy.x$ and $C[u] =_{\beta\eta} \lambda xy.y$.
- Separation theorem in ludics: for any (atomic) cut-free N, M, $N = M$ iff $P \perp N \iff P \perp M$ for any P
- Strong separation for finite data designs:

  for any finite data design d, there is a counter design d^c such that for any e,

$$d = e \quad \text{iff} \quad d^c \perp e.$$

- "For any $w \in \Sigma^*$ there is a DFA M such that $L(M) = \{w\}$."
- How do we separate an arbitrary set of data designs?

Strong separation for data sets

- Strong separation for sets of finite data designs:

  for any set $\mathbf{D}$ of finite data designs, there is a counter design $\mathbf{D}^c$ s.t. $e \in \mathbf{D}$ iff $\mathbf{D}^c \perp e$.

Strong separation for data sets

- Strong separation for sets of finite data designs:
  for any set $\mathbf{D}$ of finite data designs, there is a counter design $\mathbf{D}^c$ s.t. $e \in \mathbf{D}$ iff $\mathbf{D}^c \perp e$.
- $\mathbf{D}^c = \sum\{d^c : d \in \mathbf{D}\}$.

Strong separation for data sets

- Strong separation for sets of finite data designs:
  for any set $\mathbf{D}$ of finite data designs, there is a counter design $\mathbf{D}^c$ s.t. $e \in \mathbf{D}$ iff $\mathbf{D}^c \perp e$.
- $\mathbf{D}^c = \sum\{d^c : d \in \mathbf{D}\}$.
- Linearity in linear logic: $f(a+b) = f(a) + f(b)$

Strong separation for data sets

- Strong separation for sets of finite data designs:
  for any set $\mathbf{D}$ of finite data designs, there is a counter design $\mathbf{D}^c$ s.t. $e \in \mathbf{D}$ iff $\mathbf{D}^c \perp e$.
- $\mathbf{D}^c = \sum\{d^c : d \in \mathbf{D}\}$.
- Linearity in linear logic: $f(a+b) = f(a) + f(b)$
- Linearity in ludics: $[\![(\sum P_i)[N/x_0]]\!] = \sum([\![P_i[N/x_0]]\!])$

$$\begin{aligned}
\mathbf{D}^c \perp e \quad & \Longleftrightarrow \quad [\![\mathbf{D}^c[e/x_0]]\!] = \maltese \\
& \Longleftrightarrow \quad [\![d^c[e/x_0]]\!] = \maltese \text{ for some } d \in \mathbf{D} \\
& \Longleftrightarrow \quad d^c \perp e \text{ for some } d \in \mathbf{D} \\
& \Longleftrightarrow \quad e \in \mathbf{D}.
\end{aligned}$$

Strong separation for data sets

- Strong separation for sets of finite data designs:
  for any set $\mathbf{D}$ of finite data designs, there is a counter design $\mathbf{D}^c$ s.t. $e \in \mathbf{D}$ iff $\mathbf{D}^c \perp e$.
- $\mathbf{D}^c = \sum\{d^c : d \in \mathbf{D}\}$.
- Linearity in linear logic: $f(a+b) = f(a) + f(b)$
- Linearity in ludics: $[\![(\sum P_i)[N/x_0]]\!] = \sum([\![P_i[N/x_0]]\!])$

$$\begin{aligned} \mathbf{D}^c \perp e \quad & \Longleftrightarrow \quad [\![\mathbf{D}^c[e/x_0]]\!] = \maltese \\ & \Longleftrightarrow \quad [\![d^c[e/x_0]]\!] = \maltese \text{ for some } d \in \mathbf{D} \\ & \Longleftrightarrow \quad d^c \perp e \text{ for some } d \in \mathbf{D} \\ & \Longleftrightarrow \quad e \in \mathbf{D}. \end{aligned}$$

- Behaviours are rich enough to capture all sets of finite data.

Behaviours

- Given a set $\mathbf{T}$ of l-designs (atomic, of the same polarity),

$$\mathbf{T}^{\perp} = \{U : \forall T \in \mathbf{T}. T \perp U\}.$$

- Forms a Galois connection:

$$\mathbf{P} \subseteq \mathbf{N}^{\perp} \iff \mathbf{N} \subseteq \mathbf{P}^{\perp}$$

- Behaviour: $\mathbf{T} = \mathbf{T}^{\perp\perp}$.
- Analogue of formulas, types, computability predicates, and languages.
- Fact: Any set of the form $\{P\}^{\perp}$ is a behaviour.

Behaviours

- Any set $\mathbf{D}$ of finite data designs ‘forms’ a behaviour $\mathbf{D} \doteq \{\mathbf{D}^c\}^\perp$
- Any r.e. set $\mathbf{L} \subseteq \Sigma^*$ can be expressed as $\{P\}^\perp$ where P is finitely generated.
- Any regular set $\mathbf{L} \subseteq \Sigma^*$ can be expressed as $\{P\}^\perp$ where P is finitely generated and cut-free.
- One can apply logical connectives to obtain a new behaviour.

Outline

1. Time and space sensitive compositions in lambda calculus
2. What is ludics?
3. Data and computation in ludics
4. Arbitrary data sets
5. Language operators and internal completeness
6. Space compression and focalization
7. Conclusion

Interaction and Construction

- Two approaches to define a language

Interaction and Construction

- Two approaches to define a language
  - By interaction: $L = L(M)$ for a machine/automaton M.

Interaction and Construction

- Two approaches to define a language
  - By interaction: $L = L(M)$ for a machine/automaton M.
  - By construction: $L_1 \cup L_2,\ L_1^*$, etc.

Interaction and Construction

- Two approaches to define a language
  - By interaction: $L = L(M)$ for a machine/automaton M.
  - By construction: $L_1 \cup L_2,\ L_1^*$, etc.
- To define a behaviour

Interaction and Construction

- Two approaches to define a language
  - By interaction: $L = L(M)$ for a machine/automaton M.
  - By construction: $L_1 \cup L_2,\ L_1^*$, etc.
- To define a behaviour
  - By interaction: $\{P\}^{\perp}$ for an l-design P.

Interaction and Construction

- Two approaches to define a language
  - By interaction: $L = L(M)$ for a machine/automaton M.
  - By construction: $L_1 \cup L_2,\ L_1^*$, etc.
- To define a behaviour
  - By interaction: $\{P\}^{\perp}$ for an l-design P.
  - By construction: $a.\mathbf{D} \cup b.\mathbf{E}$

Internal completeness

- Harmony of two approaches is ensured by internal completeness:

$$a.\mathbf{D} \cup b.\mathbf{E} \doteq (a.\mathbf{D} \cup b.\mathbf{E})^{\perp\perp}.$$

- The key step when proving full completeness theorem:

$$P \in \mathsf{T}^{\circ} \iff P \text{ interprets a proof of } \mathsf{T}.$$

- Think of the case

$$\begin{aligned} \mathsf{T} &= \mathsf{D} \oplus \mathsf{E} \\ \mathsf{T}^{\circ} &= (\iota_1.\mathsf{D} \cup \iota_2.\mathsf{E})^{\perp\perp} \end{aligned}$$

- Also a key to DFAs = Regular Expressions.

Outline

1. Time and space sensitive compositions in lambda calculus
2. What is ludics?
3. Data and computation in ludics
4. Arbitrary data sets
5. Language operators and internal completeness
6. Space compression and focalization
7. Conclusion

Space compression and Focalization

- Space compression theorem: based on compression of data by using more symbols.

$$(0110)_2 \longmapsto (12)_4.$$

In terms of data designs,

$$\uparrow\overline{0}\langle\uparrow\overline{1}\langle\uparrow\overline{1}\langle\uparrow\overline{0}\langle\uparrow\overline{\mathsf{nil}}\rangle\rangle\rangle\rangle \longmapsto \uparrow\overline{1}\langle\uparrow\overline{2}\langle\uparrow\overline{\mathsf{nil}}\rangle\rangle.$$

- This map can be derived from a general principle of focalization:

$$\overline{\alpha}\langle\uparrow\overline{\beta}\langle\mathbf{N}\rangle\rangle \cong \overline{\alpha\beta}\langle\mathbf{N}\rangle.$$

Focalization principle

- In proof search in linear logic, one has to focus on a formula in the end sequent:

$$\frac{\vdash \Gamma_1, A \quad \vdash \Gamma_2, B \oplus C}{\vdash \Gamma_1, \Gamma_2, A \otimes (B \oplus C)}$$

Focalization principle

- In proof search in linear logic, one has to focus on a formula in the end sequent:

$$\frac{\vdash \Gamma_1, A \quad \vdash \Gamma_2, B \oplus C}{\vdash \Gamma_1, \Gamma_2, A \otimes (B \oplus C)}$$

- What happens next? Do we have to change the focus?

Focalization principle

- In proof search in linear logic, one has to focus on a formula in the end sequent:

$$\frac{\vdash \Gamma_1, A \quad \vdash \Gamma_2, B \oplus C}{\vdash \Gamma_1, \Gamma_2, A \otimes (B \oplus C)}$$

- What happens next? Do we have to change the focus?
- Focalization principle (Andreoli): No! You can continue with the same focus.

$$\frac{\vdash \Gamma_1, A \quad \dfrac{\vdash \Gamma_2, B}{\vdash \Gamma_2, B \oplus C}}{\vdash \Gamma_1, \Gamma_2, A \otimes (B \oplus C)} \quad \text{or} \quad \frac{\vdash \Gamma_1, A \quad \dfrac{\vdash \Gamma_2, C}{\vdash \Gamma_2, B \oplus C}}{\vdash \Gamma_1, \Gamma_2, A \otimes (B \oplus C)}$$

Focalization principle

- In proof search in linear logic, one has to focus on a formula in the end sequent:

$$\frac{\vdash \Gamma_1, A \quad \vdash \Gamma_2, B \oplus C}{\vdash \Gamma_1, \Gamma_2, A \otimes (B \oplus C)}$$

- What happens next? Do we have to change the focus?
- Focalization principle (Andreoli): No! You can continue with the same focus.

$$\frac{\vdash \Gamma_1, A \quad \dfrac{\vdash \Gamma_2, B}{\vdash \Gamma_2, B \oplus C}}{\vdash \Gamma_1, \Gamma_2, A \otimes (B \oplus C)} \quad \text{or} \quad \frac{\vdash \Gamma_1, A \quad \dfrac{\vdash \Gamma_2, C}{\vdash \Gamma_2, B \oplus C}}{\vdash \Gamma_1, \Gamma_2, A \otimes (B \oplus C)}$$

- Two connectives of the same polarity can be combined together: $A \otimes (B \oplus C) = \otimes\oplus(A, B, C)$.

Conclusion

- Ludics: general setting for logically analyzing computation
  - Supports various data, higher order, concurrency
- Importance of finite generation and cuts
  - Arbitrary l-designs: arbitrary sets of finite data
  - F.g. l-designs: r.e. languages
  - F.g. cut-free l-designs: regular languages

  “Adding stacks to automata = adding cuts to designs”
- Uses of logical theorems of ludics (separation, linearity, internal completeness, focalization)
- WIP
  - Internal/full completeness $\rightsquigarrow$ DFA = Regular Expr.
  - Focalization $\rightsquigarrow$ Space compression.