# Transductions in typed $\lambda$-calculi

Nguyễn Lê Thành Dũng (a.k.a. Tito) — nltd@nguyentito.eu
LIPN, Université Paris 13

Journées du GT ALGA, October 10th, 2019

# Some context

How to define classes of functions without machine models?

| complexity classes | automata theory |
|---|---|
| a. Descriptive Complexity | b. Monadic Second-Order Logic |

Examples:

a. *Existential* SO predicates over finite structures = NP

b. *Monadic* SO predicates over strings = *regular languages*

# Some context

How to define classes of functions without machine models?

| complexity classes | automata theory |
|---|---|
| a. Descriptive Complexity | b. Monadic Second-Order Logic |
| c. Implicit Complexity | |

*Declarative programming* vs *functional programming*. Examples:

a. *Existential* SO predicates over finite structures = NP

b. *Monadic* SO predicates over strings = *regular languages*

c. Programs in *Light Linear Logic* = P (Girard 1998)
($\cong$ more-or-less a variant of the $\lambda$-calculus)

# Some context

How to define classes of functions without machine models?

| complexity classes | automata theory |
|---|---|
| a. Descriptive Complexity | b. Monadic Second-Order Logic |
| c. Implicit Complexity | d. **our topic!** |

*Declarative programming* vs *functional programming*. Examples:

a. *Existential* SO predicates over finite structures = NP

b. *Monadic* SO predicates over strings = *regular languages*

c. Programs in *Light Linear Logic* = P (Girard 1998)
(≅ more-or-less a variant of the $\lambda$-calculus)

d. Until now, a single result on regular languages in the *simply typed $\lambda$-calculus* (Hillebrand & Kanellakis 1996)
(Somewhat related: *higher-order model checking*)

# Wait a moment, what is the $\lambda$-calculus?

A naive syntactic theory of functions:

$$\text{Terms} \quad t, u ::= x \mid t\ u \mid \lambda x.\ t \quad (x \in \text{variables})$$

$$\begin{array}{rcl} f\,x & \approx & f(x) \\ \lambda x.\ t & \approx & x \mapsto t \\ (\lambda x.\ t)\ u =_\beta t\{x := u\} & \approx & (x \mapsto x^2 + 1)(42) = 42^2 + 1 \end{array}$$

# Wait a moment, what is the $\lambda$-calculus?

A naive syntactic theory of functions:

$$\text{Terms} \quad t, u ::= x \mid t\ u \mid \lambda x.\ t \quad (x \in \text{variables})$$

$$\begin{array}{rcl} f\,x & \approx & f(x) \\ \lambda x.\ t & \approx & x \mapsto t \\ (\lambda x.\ t)\ u =_\beta t\{x := u\} & \approx & (x \mapsto x^2 + 1)(42) = 42^2 + 1 \end{array}$$

*Church encodings* of strings over $\Sigma = \{a, b\}$: morally,
$w = w[1] \ldots w[n] \in \Sigma^* \rightsquigarrow \overline{w} : f \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$

$$\overline{aab} = \lambda f_a.\ (\lambda f_b.\ (\lambda x.\ f_a\ (f_a\ (f_b\ x)))) \quad \text{for } \Sigma = \{a, b\}$$

# Wait a moment, what is the $\lambda$-calculus?

A naive syntactic theory of functions:

$$\text{Terms} \quad t, u ::= x \mid t\ u \mid \lambda x.\ t \quad (x \in \text{variables})$$

$$\begin{array}{rcl} f\,x & \approx & f(x) \\ \lambda x.\ t & \approx & x \mapsto t \\ (\lambda x.\ t)\ u =_\beta t\{x := u\} & \approx & (x \mapsto x^2 + 1)(42) = 42^2 + 1 \end{array}$$

*Church encodings* of strings over $\Sigma = \{a, b\}$: morally,
$w = w[1] \ldots w[n] \in \Sigma^* \rightsquigarrow \overline{w} : f \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$

$$\overline{aab} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_a\ (f_b\ x)) \quad \text{for } \Sigma = \{a, b\}$$

# The simply typed $\lambda$-calculus

Add a *type system*: specifications for $\lambda$-terms

$$t : A \to B \quad \approx \quad \text{“}t \text{ is a function from } A \text{ to } B\text{”}$$

*Simple types*: $A, B ::= o \mid A \to B$

**Example**

If $f_a : o \to o$ and $f_b : o \to o$ then $\lambda x.\, f_a\ (f_a\ (f_b\ x))) : o \to o$.

# The simply typed $\lambda$-calculus

Add a *type system*: specifications for $\lambda$-terms

$$t : A \to B \quad \approx \quad \text{“}t \text{ is a function from } A \text{ to } B\text{”}$$

*Simple types*: $A, B ::= o \mid A \to B$

**Example**

If $f_a : o \to o$ and $f_b : o \to o$ then $\lambda x.\, f_a\ (f_a\ (f_b\ x))) : o \to o$.

**Proposition**

*There exists a type* $\mathtt{Str}_\Sigma$ *such that* $\forall w \in \Sigma^*.\ \overline{w} : \mathtt{Str}_\Sigma$,
*and conversely* $\forall t.\ (t : \mathtt{Str}_\Sigma \implies \exists w \in \Sigma^*.\ t =_\beta \overline{w})$.

Moreover, $\overline{w} : \mathtt{Str}_\Sigma\{o := A\}$ for any type $A$.

# Regular languages in the simply typed $\lambda$-calculus

**Theorem (Hillebrand & Kanellakis 1996)**

$L \subseteq \Sigma^*$ *is a* regular language *iff*
*there exist a simple type* $A$ *and a term* $t : \mathtt{Str}_\Sigma\{o := A\} \to \mathtt{Bool}$
*such that* $\forall w \in \Sigma^*,\ (t\ \overline{w} =_\beta \mathtt{true} \iff w \in L)$.

(Proof by semantics, in backup slide)

# Regular languages in the simply typed $\lambda$-calculus

**Theorem (Hillebrand & Kanellakis 1996)**

$L \subseteq \Sigma^*$ *is a* regular language *iff*
*there exist a simple type* $A$ *and a term* $t : \mathtt{Str}_\Sigma\{o := A\} \to \mathtt{Bool}$
*such that* $\forall w \in \Sigma^*,\ (t\ \overline{w} =_\beta \mathtt{true} \iff w \in L)$.

(Proof by semantics, in backup slide)

**Open question**

What about the *functions* $\Gamma^* \to \Sigma^*$ definable by terms of type $\mathtt{Str}_\Gamma\{o := A\} \to \mathtt{Str}_\Sigma$ for some $A$?

When $|\Sigma| = 1$, $\mathtt{Str}_\Sigma$ becomes the canonical type of *integers*.
No known characterization of the functions $\mathbb{N} \to \mathbb{N}$
definable in the simply typed $\lambda$-calculus!

When $A = o$ see [Schwichtenberg 1975] ($\mathbb{N} \to \mathbb{N}$), [Zaionc 1987] ($\Gamma^* \to \Sigma^*$)

# Regular languages in the simply typed $\lambda$-calculus

**Theorem (Hillebrand & Kanellakis 1996)**

$L \subseteq \Sigma^*$ *is a* regular language *iff*
*there exist a simple type* $A$ *and a term* $t : \mathtt{Str}_\Sigma\{o := A\} \to \mathtt{Bool}$
*such that* $\forall w \in \Sigma^*,\ (t\ \overline{w} =_\beta \mathtt{true} \iff w \in L)$.

(Proof by semantics, in backup slide)

**Open question**

What about the *functions* $\Gamma^* \to \Sigma^*$ definable by terms of type $\mathtt{Str}_\Gamma\{o := A\} \to \mathtt{Str}_\Sigma$ for some $A$?

Corollary of above theorem: if $f : \Gamma^* \to \Sigma^*$ is such a function, then for all $L \subseteq \Sigma^*$, $L$ regular $\implies f^{-1}(L)$ regular.

# Regular languages in the simply typed $\lambda$-calculus

**Theorem (Hillebrand & Kanellakis 1996)**

$L \subseteq \Sigma^*$ *is a* regular language *iff*
*there exist a simple type* $A$ *and a term* $t : \mathtt{Str}_\Sigma\{o := A\} \to \mathtt{Bool}$
*such that* $\forall w \in \Sigma^*,\ (t\ \overline{w} =_\beta \mathtt{true} \iff w \in L)$.

(Proof by semantics, in backup slide)

**Open question**

What about the *functions* $\Gamma^* \to \Sigma^*$ definable by terms of type $\mathtt{Str}_\Gamma\{o := A\} \to \mathtt{Str}_\Sigma$ for some $A$?

Corollary of above theorem: if $f : \Gamma^* \to \Sigma^*$ is such a function, then for all $L \subseteq \Sigma^*$, $L$ regular $\implies f^{-1}(L)$ regular.

We should look at *transducers*, i.e. automata with output.

# Regular languages in the simply typed $\lambda$-calculus

**Theorem (Hillebrand & Kanellakis 1996)**

$L \subseteq \Sigma^*$ *is a* regular language *iff*
*there exist a simple type* $A$ *and a term* $t : \mathtt{Str}_\Sigma\{o := A\} \to \mathtt{Bool}$
*such that* $\forall w \in \Sigma^*,\ (t\ \overline{w} =_\beta \mathtt{true} \iff w \in L)$.

(Proof by semantics, in backup slide)

**Open question**

What about the *functions* $\Gamma^* \to \Sigma^*$ definable by terms of type $\mathtt{Str}_\Gamma\{o := A\} \to \mathtt{Str}_\Sigma$ for some $A$?

Corollary of above theorem: if $f : \Gamma^* \to \Sigma^*$ is such a function, then for all $L \subseteq \Sigma^*$, $L$ regular $\implies f^{-1}(L)$ regular.

We should look at *transducers*, i.e. automata with output.
Even then, the problem stays hard. Let's see an easier one.

# A simplifying assumption: linearity

We add *linear types*[1]: $A, B ::= o \mid A \multimap B \mid A \to B$

- $A \multimap B$: *linear* functions "using their argument *only once*" e.g. $\lambda x.\, t$ where $t$ has a single occurrence of $x$
- $A \to B$: usual *non-linear* functions

What are the functions definable by terms of type $\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma$ with $A$ *purely linear* ('$\to$'-free)?

[1] Design choices taken from Clairambault et al. POPL'17

# A simplifying assumption: linearity

We add *linear types*[1]: $A, B ::= o \mid A \,\&\, B \mid A \multimap B \mid A \to B$

- $A \multimap B$: *linear* functions "using their argument *only once*" e.g. $\lambda x.\, t$ where $t$ has a single occurrence of $x$
- $A \to B$: usual *non-linear* functions
- $A \,\&\, B$: so-called "additive" Cartesian product
  $\lambda x.\, \langle x, x \rangle : A \multimap (A \,\&\, A)$ is linear, but $\lambda p.\, \pi_1(p)\, \pi_2(p)$ is not used to get enough expressivity

What are the functions definable by terms of type $\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma$ with $A$ *purely linear* ('$\to$'-free)?

[1] Design choices taken from Clairambault et al. POPL'17

# A simplifying assumption: linearity

We add *linear types*[1]: $A, B ::= o \mid A \,\&\, B \mid A \multimap B \mid A \to B$

- $A \multimap B$: *linear* functions "using their argument *only once*"
  e.g. $\lambda x.\, t$ where $t$ has a single occurrence of $x$
- $A \to B$: usual *non-linear* functions
- $A \,\&\, B$: so-called "additive" Cartesian product
  $\lambda x.\, \langle x, x \rangle : A \multimap (A \,\&\, A)$ is linear, but $\lambda p.\, \pi_1(p)\, \pi_2(p)$ is not
  used to get enough expressivity
  $\longrightarrow$ That is, to characterize a known class of transductions.

What are the functions definable by terms of type
$\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma$ with $A$ *purely linear* ('$\to$'-free)?

[1] Design choices taken from Clairambault et al. POPL'17

# Characterizing regular functions

**Theorem**

*$f : \Gamma^* \to \Sigma^*$ is a* regular function *iff there exist*
*a* purely linear *type $A$ and a term $t : \mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma$*
*such that $\forall w \in \Sigma^*,\ t\ \overline{w} =_\beta \overline{f(w)}$.*

Equivalent characterizations of regular functions include:

- MSO-definable transformations *à la* Courcelle
- copyless streaming string transducers
- *two-way* finite state transducers
- basic functions + combinators (at least 3 possible variants)

# Characterizing regular functions

**Theorem**

*$f : \Gamma^* \to \Sigma^*$ is a* regular function *iff there exist*
*a* purely linear *type $A$ and a term $t : \mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma$*
*such that $\forall w \in \Sigma^*,\ t\ \overline{w} =_\beta \overline{f(w)}$.*

Equivalent characterizations of regular functions include:

- MSO-definable transformations *à la* Courcelle
- *copyless* (i.e. linear) streaming string transducers
- *two-way* finite state transducers
- basic functions + combinators (at least 3 possible variants)

# Characterizing regular functions

**Theorem**

*f is a* regular function *between* ranked trees *iff there exist a* purely linear *type* $A$ *and a term* $t : \mathtt{Tree}_\Gamma\{o := A\} \multimap \mathtt{Tree}_\Sigma$ ($\Gamma$, $\Sigma$ *ranked alphabets*) *such that* $\forall w,\ t\ \overline{w} =_\beta \overline{f(w)}$.

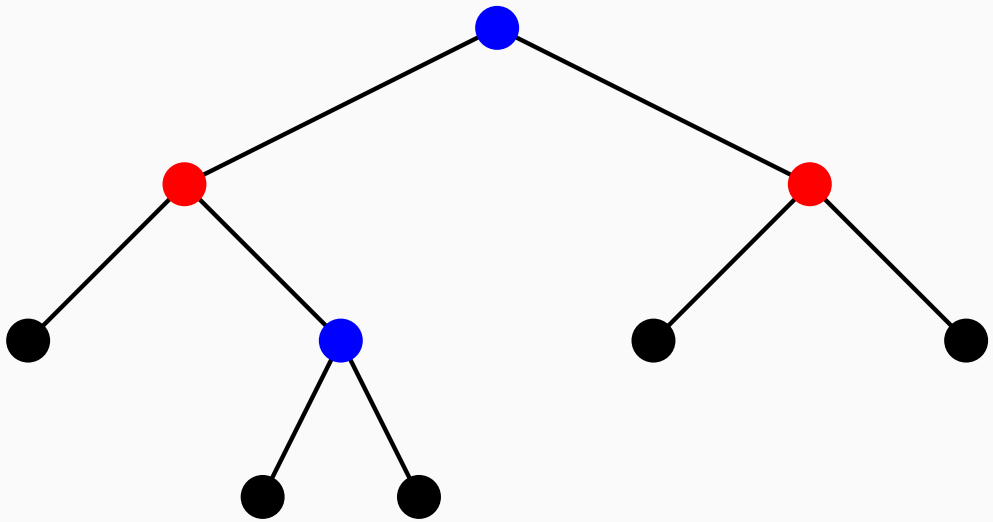
Extends without difficulty to Church-encoded trees!

Equivalent characterizations of regular tree functions include:

- MSO-definable transformations *à la* Courcelle
- "single use restricted" streaming tree transducers

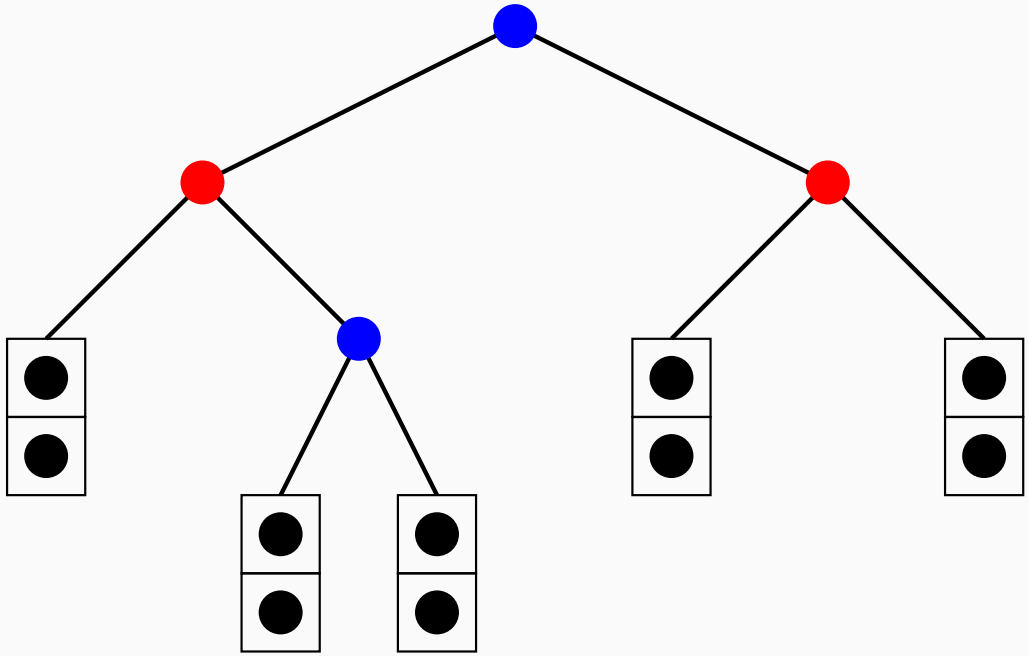
# Bottom-up streaming tree transducer: example

$f : T \mapsto \bullet(T, \mathsf{reverse}(T))$



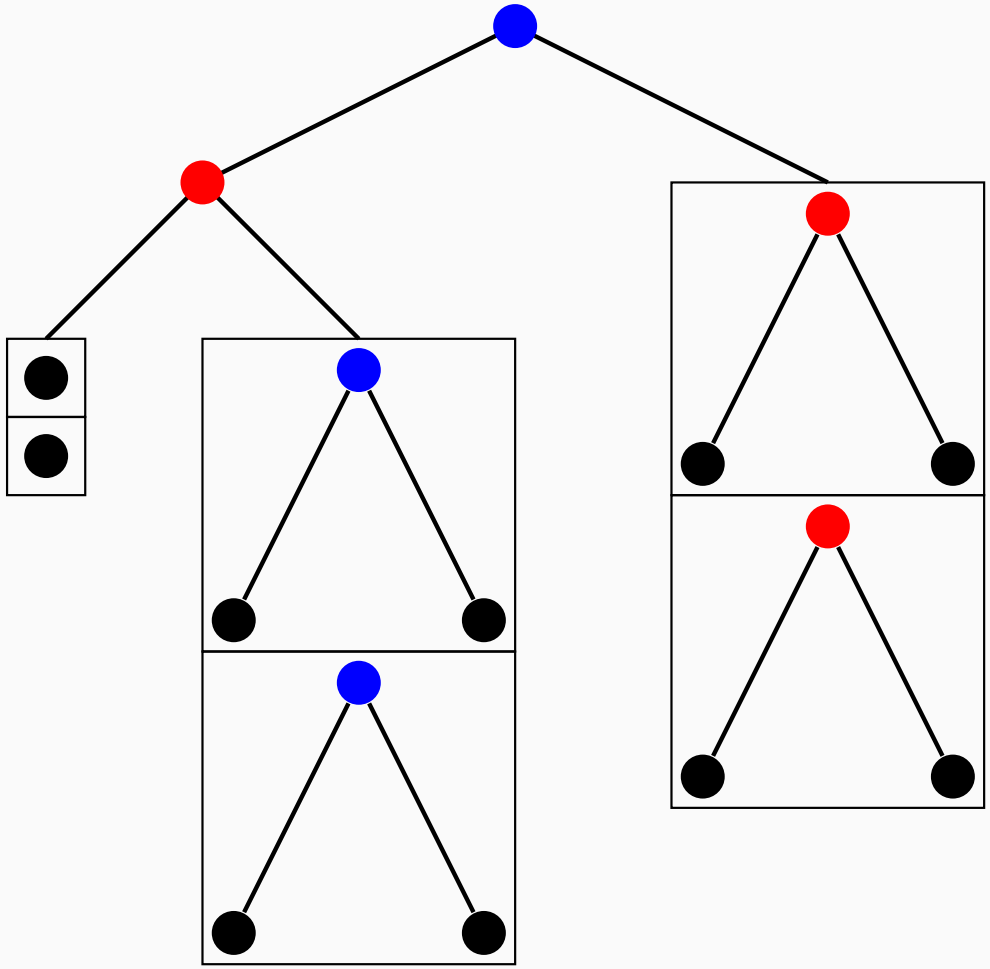
# Bottom-up streaming tree transducer: example

$f : T \mapsto \bullet(T, \mathtt{reverse}(T)) = \bullet(\text{top register}, \text{bottom register})$



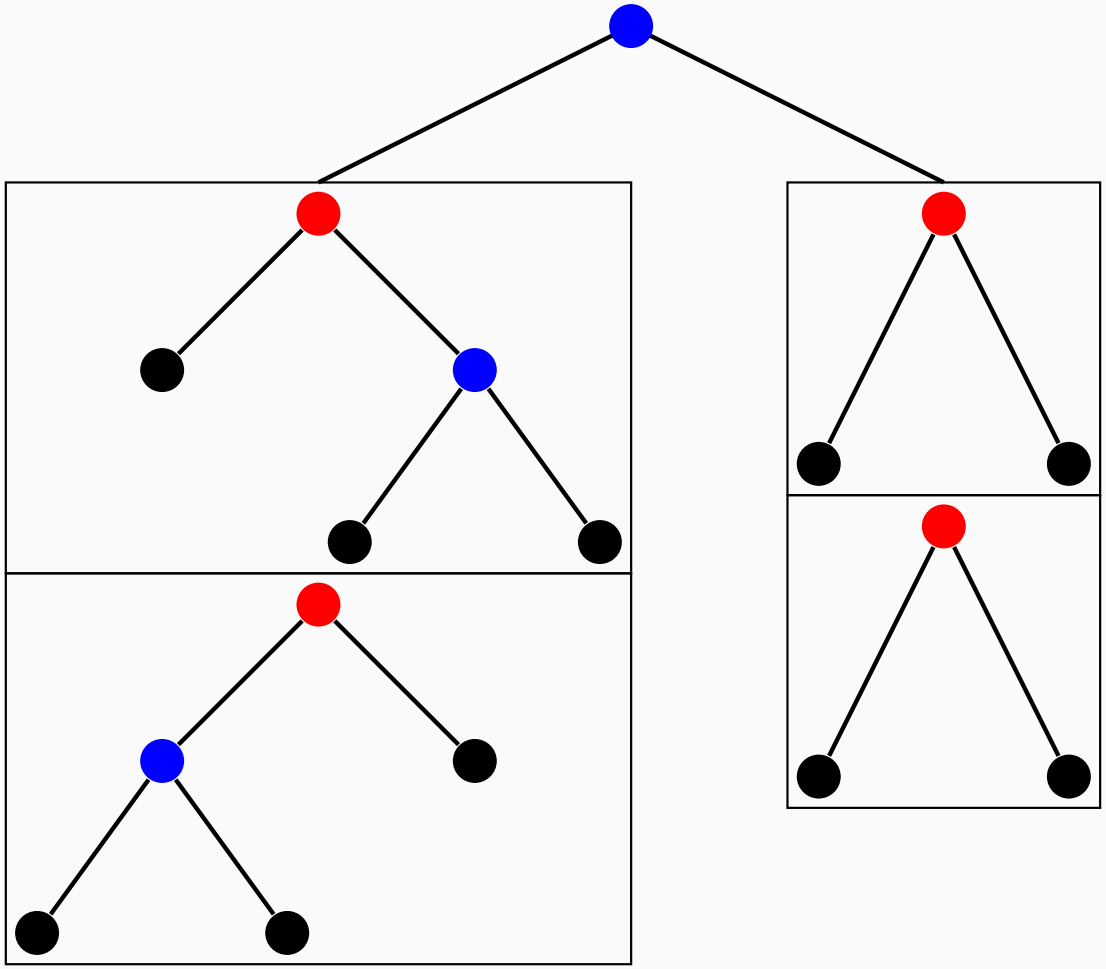
# Bottom-up streaming tree transducer: example

$f : T \mapsto \bullet(T, \mathtt{reverse}(T)) = \bullet(\text{top register}, \text{bottom register})$



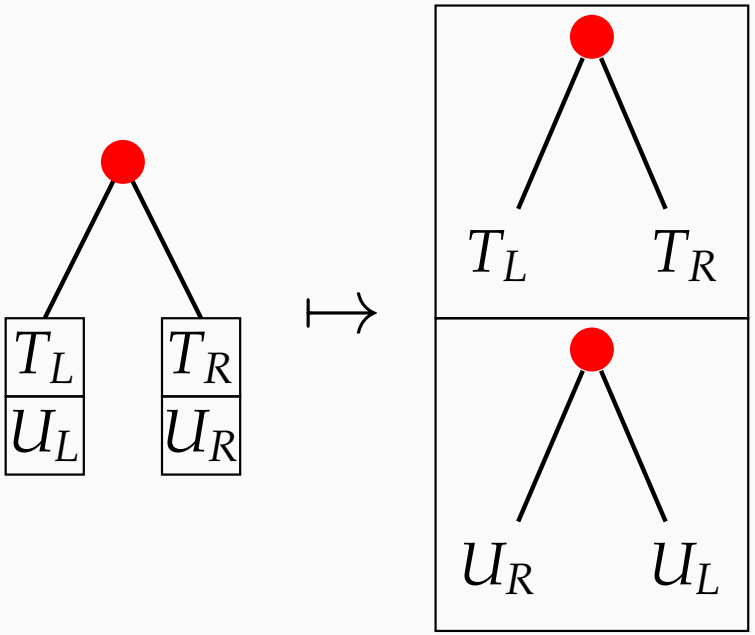
# Bottom-up streaming tree transducer: example

$f : T \mapsto \bullet(T, \mathtt{reverse}(T)) = \bullet(\text{top register}, \text{bottom register})$



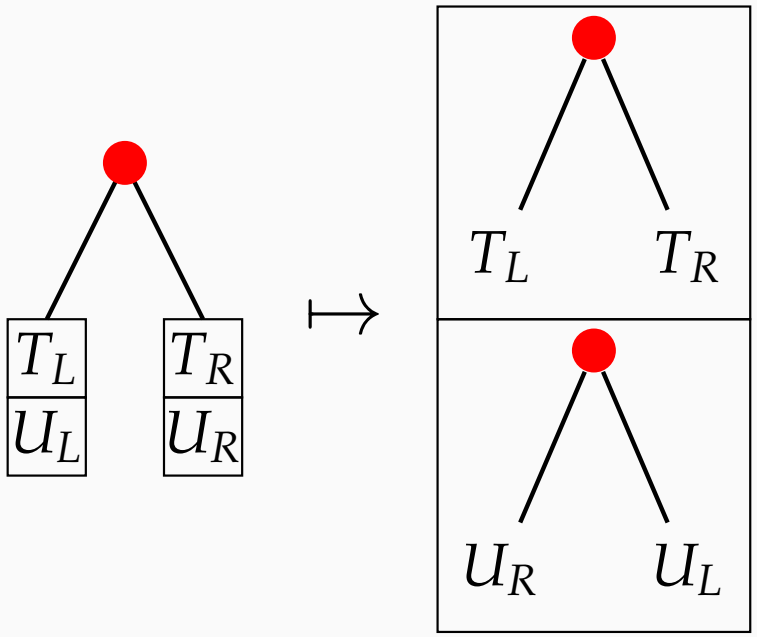
# A copyless streaming tree transducer

Copyless assignment: each register from children used *at most once* in parent's registers



# A copyless streaming tree transducer

Copyless assignment: each register from children used *at most once* in parent's registers



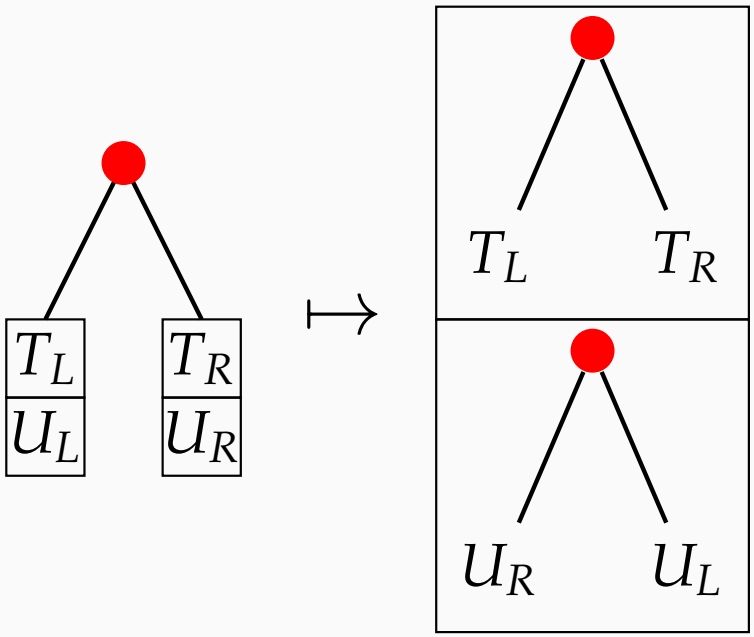

i.e. each variable on left appears at most once on right

Compare *linear* function $\lambda x.\, t$ where $x$ appears at most once in $t$

# A single-use-restricted streaming tree transducer

Another example: top register = subtree, bottom = $f$(subtree)

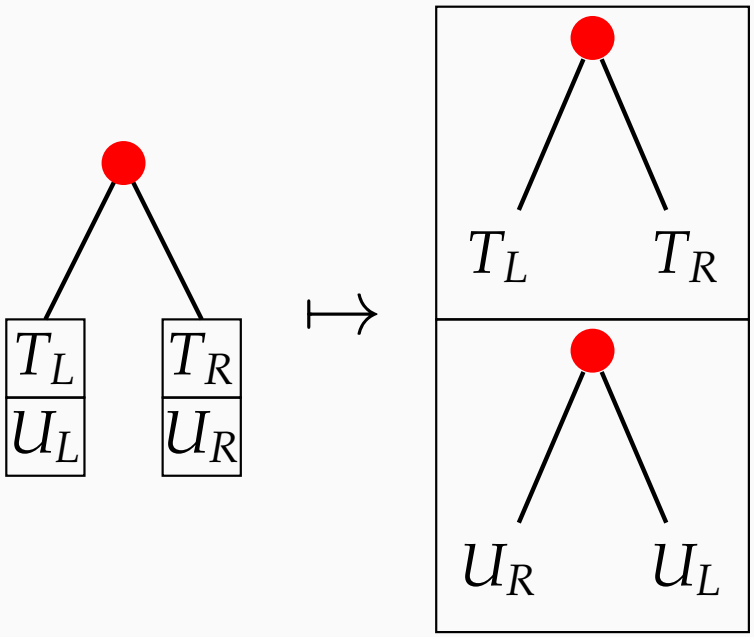
$$f(\bullet(U_L, U_R)) = \bullet(f(U_R), f(U_L))$$



# A single-use-restricted streaming tree transducer

Another example: top register = subtree, bottom = $f$(subtree)

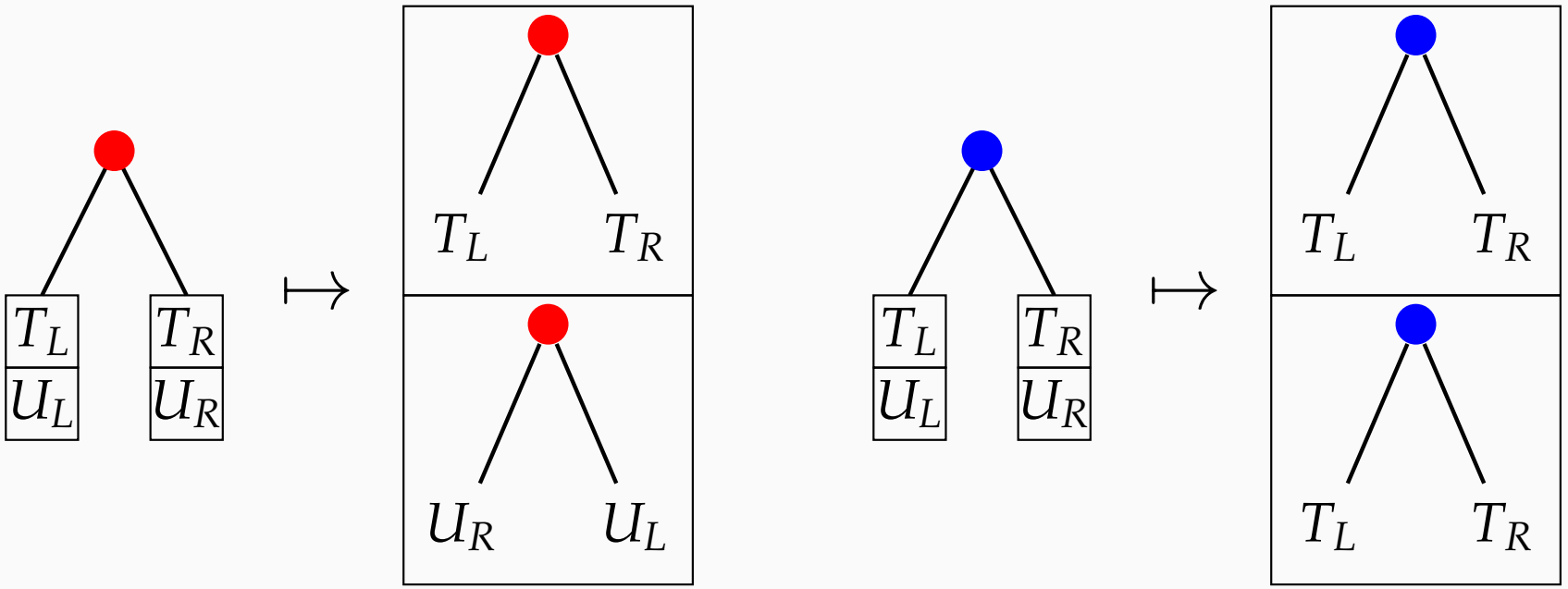
$$f(\bullet(U_L, U_R)) = \bullet(f(U_R), f(U_L)) \qquad f(T) = T \text{ otherwise}$$



# A single-use-restricted streaming tree transducer

Another example: top register = subtree, bottom = $f$(subtree)

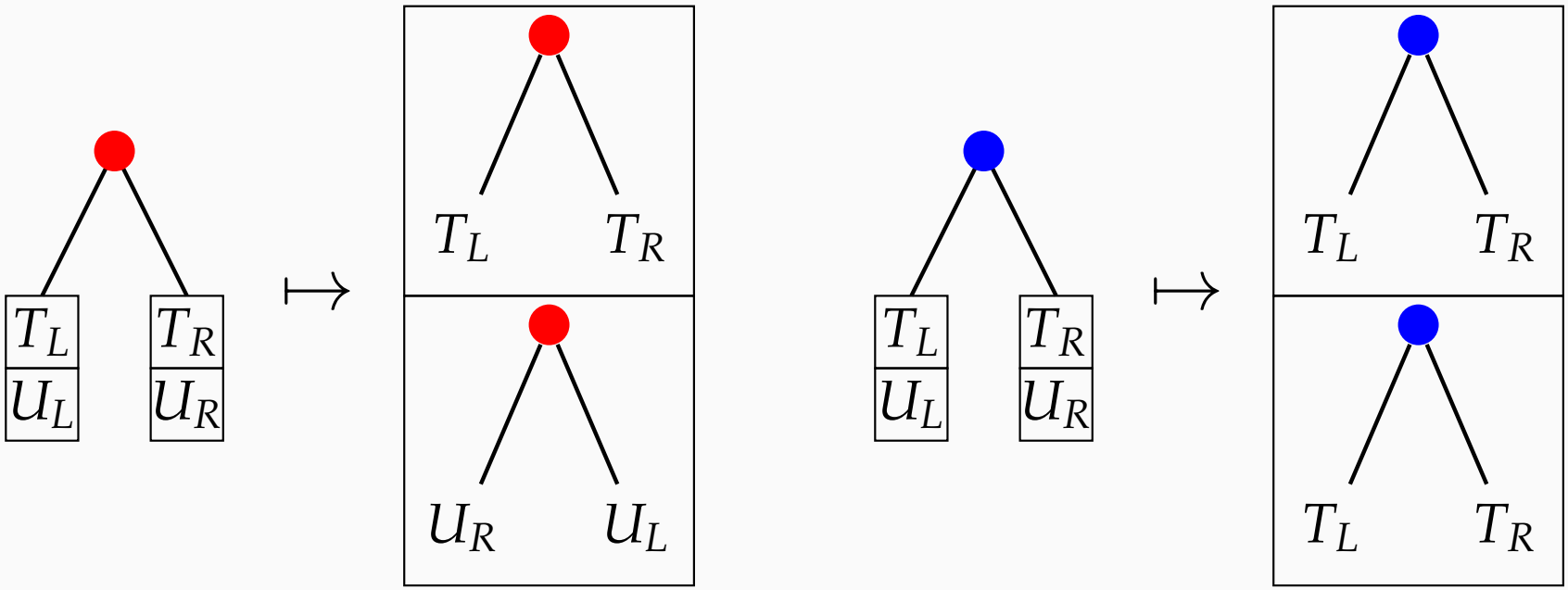
$$f(\bullet(U_L, U_R)) = \bullet(f(U_R), f(U_L)) \qquad f(T) = T \text{ otherwise}$$



# A single-use-restricted streaming tree transducer

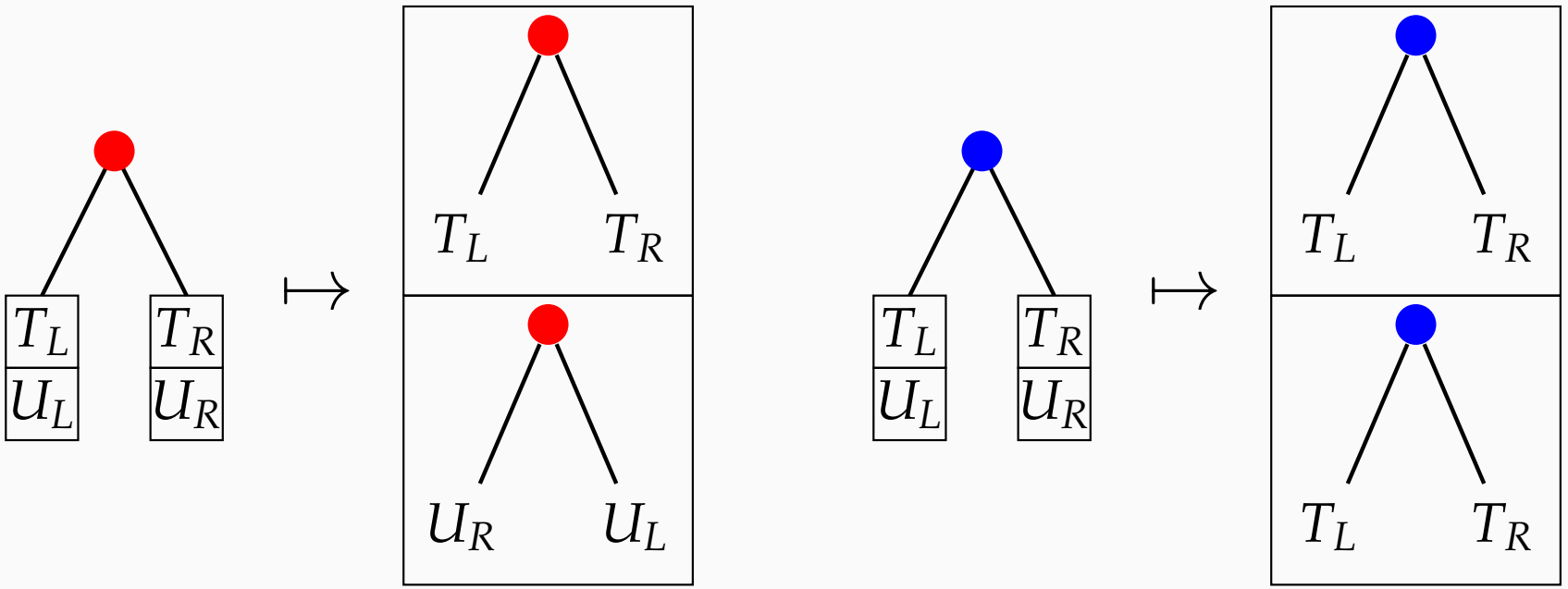
Another example: top register = subtree, bottom = $f$(subtree)

$$f(\bullet(U_L, U_R)) = \bullet(f(U_R), f(U_L)) \qquad f(T) = T \text{ otherwise}$$



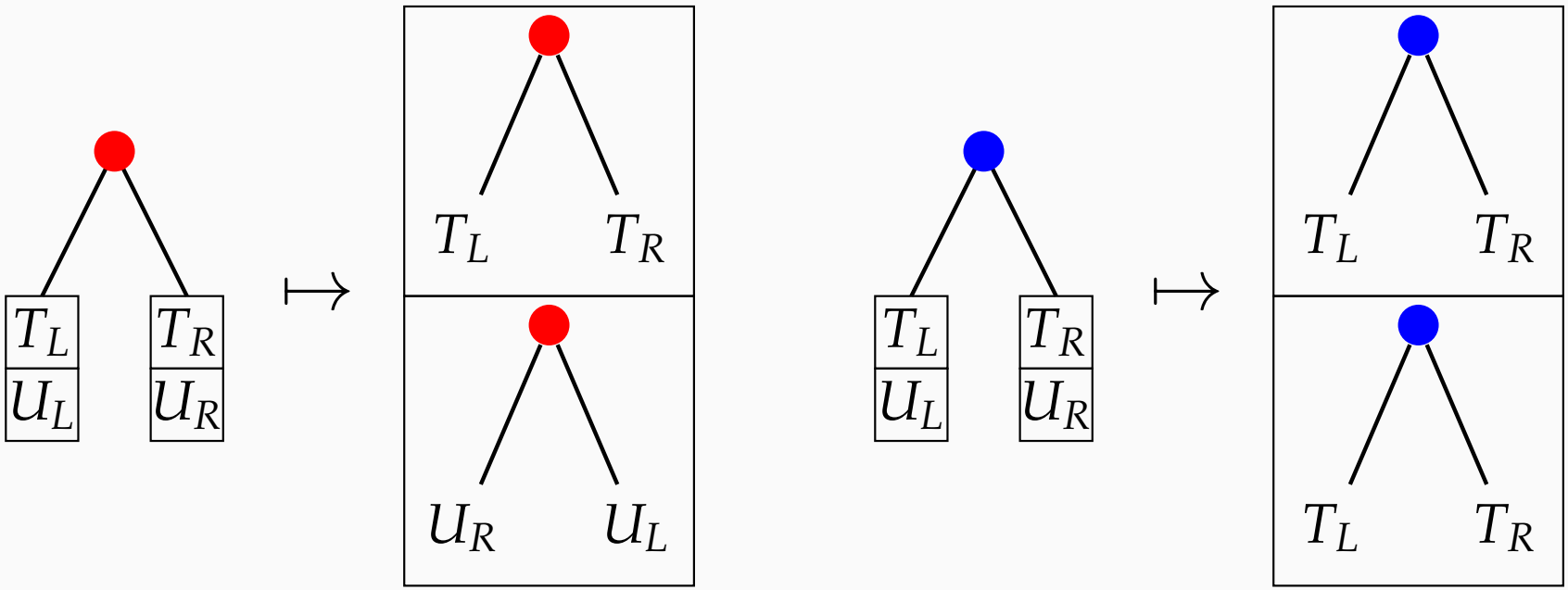

*Not copyless*! A copyless implementation might be impossible...
But this function is regular (i.e. MSO-definable).

# A single-use-restricted streaming tree transducer




Fix: declare the 2 registers to be *conflicting*

# A single-use-restricted streaming tree transducer




Fix: declare the 2 registers to be *conflicting*

$\longrightarrow$ The "single use restriction" of streaming tree transducers depends on a *conflict relation* on registers

# Single use restriction vs linear logic

The "single use restriction" of streaming tree transducers depends on a (reflexive, symmetric) *conflict relation* on registers

A child register can be used in 2 conflicting parent registers
$\simeq$ linearity of $\lambda x.\, \langle x, x \rangle : A \multimap A \mathbin{\&} A$

# Single use restriction vs linear logic

The "single use restriction" of streaming tree transducers depends on a (reflexive, symmetric) *conflict relation* on registers

A child register can be used in 2 conflicting parent registers
$\simeq$ linearity of $\lambda x.\, \langle x, x \rangle : A \multimap A \mathbin{\&} A$

Counterpart: $\pi_1(p)$ and $\pi_2(p)$ can't be used at the same time

# Single use restriction vs linear logic

The "single use restriction" of streaming tree transducers depends on a (reflexive, symmetric) *conflict relation* on registers

A child register can be used in 2 conflicting parent registers
$\simeq$ linearity of $\lambda x.\, \langle x, x \rangle : A \multimap A \mathbin{\&} A$

Counterpart: $\pi_1(p)$ and $\pi_2(p)$ can't be used at the same time
$\simeq$ conflicting registers cannot be recombined in the same tree

# Single use restriction vs linear logic

The "single use restriction" of streaming tree transducers depends on a (reflexive, symmetric) *conflict relation* on registers

A child register can be used in 2 conflicting parent registers
$\simeq$ linearity of $\lambda x.\, \langle x, x \rangle : A \multimap A \mathbin{\&} A$

Counterpart: $\pi_1(p)$ and $\pi_2(p)$ can't be used at the same time
$\simeq$ conflicting registers cannot be recombined in the same tree

- Conflicting registers $\simeq A \mathbin{\&} B$ (*additive* pairs)
  ('&' types also required to encode automata *states*)
- Non-conflicting registers $\simeq A \otimes B$ (*multiplicative* pairs)
  the other canonical conjunction in linear logic

# Single use restriction vs linear logic

The "single use restriction" of streaming tree transducers depends on a (reflexive, symmetric) *conflict relation* on registers

A child register can be used in 2 conflicting parent registers
$\simeq$ linearity of $\lambda x.\, \langle x, x \rangle : A \multimap A \mathbin{\&} A$

Counterpart: $\pi_1(p)$ and $\pi_2(p)$ can't be used at the same time
$\simeq$ conflicting registers cannot be recombined in the same tree

- Conflicting registers $\simeq A \mathbin{\&} B$ (*additive* pairs)
  ('&' types also required to encode automata *states*)
- Non-conflicting registers $\simeq A \otimes B$ (*multiplicative* pairs)
  the other canonical conjunction in linear logic
  Implicit in our $\lambda$-calculus with linear types:
  $\lambda x.\, (\lambda y.\, \ldots) \cong \lambda z.\, \texttt{let}\ x \otimes y = z\ \texttt{in}\ \ldots$

# Conclusion

"Implicit complexity for automata": characterizations of

- regular languages (Hillebrand & Kanellakis 1996)
- **regular tree functions (new!)**, via *linear types*

using Church encodings in typed $\lambda$-calculi.

Plenty of perspectives...

# Conclusion

"Implicit complexity for automata": characterizations of

- regular languages (Hillebrand & Kanellakis 1996)
- **regular tree functions (new!)**, via *linear types*

using Church encodings in typed $\lambda$-calculi.

Plenty of perspectives... Some partial results:

- copyful SST + composition $\subseteq$ simply typed $\lambda$-calculus
  (our open question at the start; copyful SST $\cong$ HDT0L systems)

# Conclusion

"Implicit complexity for automata": characterizations of

- regular languages (Hillebrand & Kanellakis 1996)
- **regular tree functions (new!)**, via *linear types*

using Church encodings in typed $\lambda$-calculi.

Plenty of perspectives... Some partial results:

- copyful SST + composition $\subseteq$ simply typed $\lambda$-calculus
  (our open question at the start; copyful SST $\cong$ HDT0L systems)
- polyregular functions $\subseteq$ parsimonious $\lambda$-calculus
  (a system coming from implicit complexity in linear logic
  relies on polyregular functions $=$ *tiered* SST + composition)

# Conclusion

"Implicit complexity for automata": characterizations of

- regular languages (Hillebrand & Kanellakis 1996)
- **regular tree functions (new!)**, via *linear types*

using Church encodings in typed $\lambda$-calculi.

Plenty of perspectives... Some partial results:

- copyful SST + composition $\subseteq$ simply typed $\lambda$-calculus
  (our open question at the start; copyful SST $\cong$ HDT0L systems)
- polyregular functions $\subseteq$ parsimonious $\lambda$-calculus
  (a system coming from implicit complexity in linear logic
  relies on polyregular functions $=$ *tiered* SST + composition)
- star-free languages $\supseteq$ non-commutative affine $\lambda$-calculus

# Hillebrand & Kanellakis's proof by semantic evaluation

Canonical semantics:

- choose set $S$, $[\![o]\!] = S$, $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$
- $t : A \rightsquigarrow [\![t]\!] \in [\![A]\!]$, e.g. $[\![f\ x]\!] = [\![f]\!]\,([\![x]\!])$

**Theorem (Hillebrand & Kanellakis 1996)**

$L \subseteq \Sigma^*$ *is a* regular language *iff*
*there exist a simple type* $A$ *and a term* $t : \mathtt{Str}_\Sigma\{o := A\} \to \mathtt{Bool}$
*such that* $\forall w \in \Sigma^*,\ (t\ \overline{w} =_\beta \mathtt{true} \iff w \in L)$.

**Proof sketch of** $(\Leftarrow)$**.**

- choose $S$ finite: $[\![\mathtt{Str}\{o := A\}]\!]$ is a finite monoid
- $w \mapsto [\![\overline{w}]\!]$ is a monoid morphism $\Sigma^* \to [\![\mathtt{Str}\{o := A\}]\!]$
- $[\![\overline{w}]\!]$ determines whether $w \in L$ □