Type systems as functors

An introduction to Melliès and Zeilberger's perspective and some applications

Damiano Mazza
CNRS, LIPN, Université Paris 13

COGITARE
Kick-off meeting
Villetaneuse, 28 March 2019

What's a type system?

(A) type system is a set of rules that assigns a property called type to the various constructs of a computer program, such as variables, expressions, functions or modules. These types formalize and enforce the otherwise implicit categories the programmer uses for algebraic data types, data structures, or other components (e.g. "string", "array of float", "function returning boolean"). — Wikipedia

The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program. — Luca Cardelli

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. — Benjamin Pierce

What's a type system?

(A) type system is a set of rules that assigns a property called type to the various constructs of a computer program, such as variables, expressions, functions or modules. These types formalize and enforce the otherwise implicit categories the programmer uses for algebraic data types, data structures, or other components (e.g. "string", "array of float", "function returning boolean"). — Wikipedia

The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program. — Luca Cardelli

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. — Benjamin Pierce

A type system is a functor. — Paul-André Melliès and Noam Zeilberger

A polysemy problem

- Types as language components: used to identify well-formed expressions; no need (nor sense) to consider "untyped" espressions.
- Types as predicates: used to identify sets of expressions with certain properties; every expression has a meaning, types assert properties of such meanings.
- For programming languages theorists:
  - "Church-style" types;
  - "Curry-style" types.

Church-style types

- Naive analogy:

  type system $\sim$ category

  A program M of type B with a parameter x of type A is an arrow

$$A \xrightarrow{M[x^A]^B} B$$

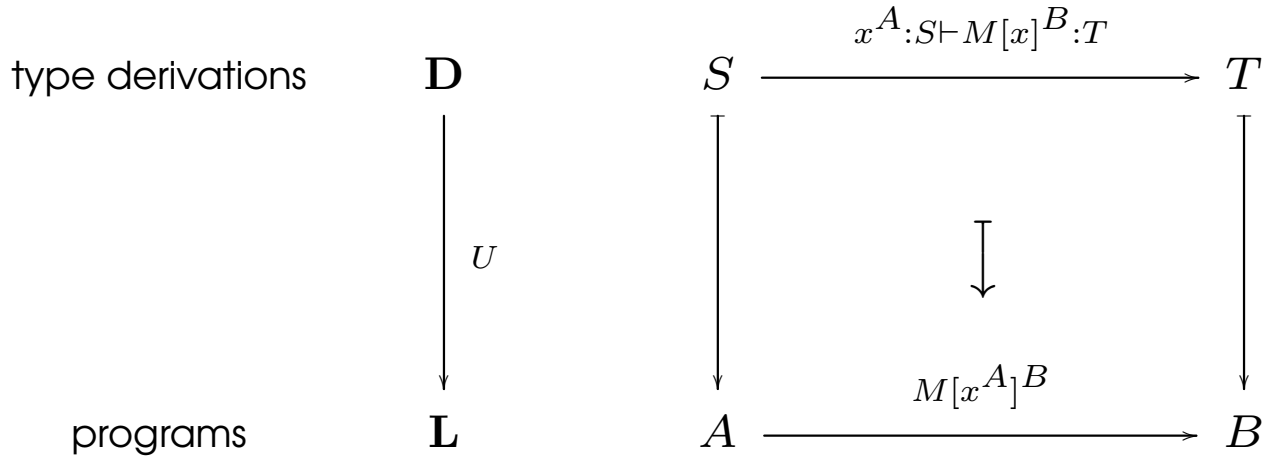
- Problematic in general:

$$\frac{x : A \vdash M : B \qquad B \leq C}{x : A \vdash M : C}\ \text{subtyping} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash M : B}{\Gamma \vdash M : A \cap B}\ \text{intersection}$$

  but an arrow in a category cannot have several targets...

Functors are Type Refinement Systems (Melliès and Zeilberger, POPL 2015)

- Instead, we consider a functor



- This is a type refinement system:
  - $U(S) = A$ means that S *refines* A;
  - $U(S \xrightarrow{\delta} T) = A \xrightarrow{M} B$ means that δ is a derivation for M;
  - $U(S \xrightarrow{\delta} T) = \mathrm{id}_A$ means that S is a subtype of T.

Example

- Let Λ be the following monoid (category with only one object):
  - elements: λ-terms with at most one free variable x;
  - operation: $M \bullet N := M[N/x]$ (substitution); identity: x.
- Let $\mathbf{STLC}$ be the following category:
  - objects: simple types $A, B ::= \alpha \mid A \to B$;
  - arrows: Church-style simply-typed λ-terms, with at most one free variable: $A \overset{M[x^A]^B}{\longrightarrow} B$;
  - composition $=$ substitution; $\mathrm{id}_A = x^A$.
- We have an obvious forgetful functor

$$\begin{array}{c} \mathbf{STLC} \\ \big\downarrow U \\ \Lambda \end{array}$$

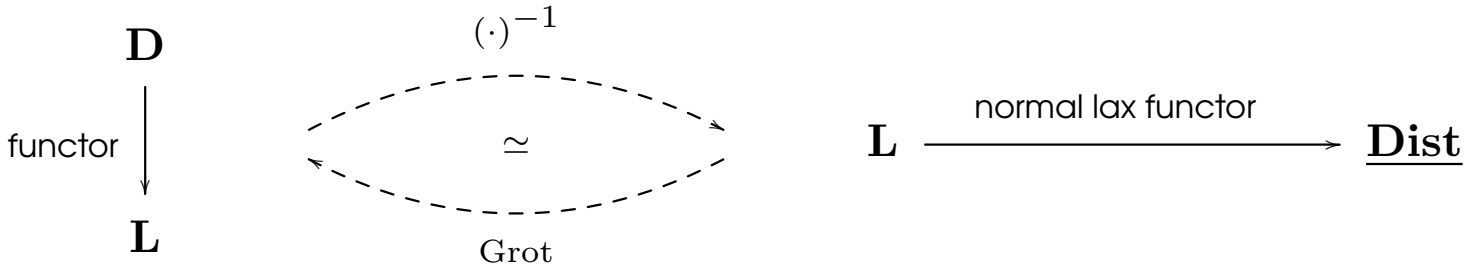
- Curry-Howard isomorphism: $\mathbf{NJ} \cong \mathbf{STLC}$. Induces a functor $\mathbf{NJ} \to \Lambda$.

More examples

- In general, when $\mathbf{L}$ has only one object, we have a type system in the usual sense.
- Hoare logic:
  - $\mathbf{L} =$ monoid of commands C (imperative language);
  - $\mathbf{D}$ category with objects $=$ predicates $\varphi, \psi \ldots$ and morphisms $=$ proofs of Hoare triples $\{\varphi\}C\{\psi\}$;
  - catesian lifting/oplifting = weakest/strongest pre/postcondition.
- Separation logic: separation connectives are also explained in terms of cartesian liftings/opliftings.
- Reynolds's (2000) coherence theorem.
- Moral: the theory of functors and the theory of types may be formulated one in the other!

Alternative viewpoint

- Type (refinement) systems are also "presheaves".
- Grothendieck construction (generalized by Bénabou):



- $\underline{\mathbf{Dist}}$ is the (pseudo) double category of distributors:
  - objects: small categories;
  - vertical arrows: functors;
  - horizontal arrows: distributors (functors $\mathbf{A}^{\mathrm{op}} \times \mathbf{B} \to \mathbf{Set}$);
  - 2-cells: certain natural trasformations.

The dimensional ladder

5. …
4. residue equivalence (Melliès's HDR)
3. standardization
2. cut-elimination/execution
1. proofs/programs
0. formulas/types

Programming languages as 2-operads

- We may reformulate Melliès and Zeilberger's approach
  - with a further dimension (2-categories);
  - in a multicategorical framework (several sources, one target)

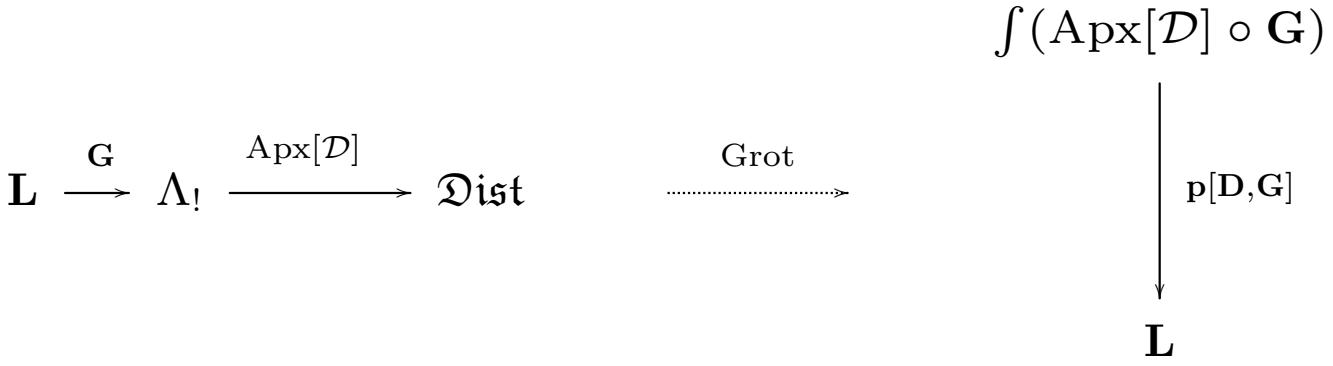
- 2-operad = *symmetric multicategory enriched on categories*
  0. objects (types);
  1. multimorphisms $A_1, \ldots, A_n \xrightarrow{M} B$
     (programs $A_1 : x_1, \ldots, A_n : x_n \vdash M : B$);
  2. 2-arrows $M \xRightarrow{\rho} N$, provided that M, N have same source and target
     (reductions/evaluation paths).

- Type (refinement) system $=$ morphism of 2-operads.

- Subject reduction/expansion $=$ oplifting/lifting of 2-arrows.

Intersection types as approximations

- Approximations in linear logic: $!A = \lim_{n\to\infty} \overbrace{(A \,\&\, 1) \otimes \cdots \otimes (A \,\&\, 1)}^{n}$
- This induces a lax morphism of 2-operads $\mathrm{Apx}[\mathcal{D}] : \Lambda_! \longrightarrow \mathfrak{Dist}$, so



- Most well-known intersection type systems arise in this way (varying $\mathcal{D}$ and $\mathbf{G}$, with $\mathbf{L} = \Lambda$).

. . . and beyond

The point of these observations is not the reduction of the familiar to the unfamiliar (. . .) but the extension of the familiar to cover many more cases. — Saunders MacLane

- Not merely explain but also predict.
- Intersection types for call-by-value λ-calculus.
- Intersection types for the $\lambda\mu$-calculus.
- Intersection types for a fragment of the π-calcul characterizing absence of runtime errors (*e.g.*deadlocks).
- "Intersection types" for "Turing machines" which may be used to prove the Cook-Levin theorem (SAT is NP-complete).
- Generalization of constraint satisfaction problems (sheaves on certain sites).