# The Types Who Say '\ni'

Conor McBride

July 17, 2017

## 1 Introduction

This paper documents a formalization of the basic metatheory for a bidirectional presentation of Martin-Löf's small and beautiful, but notoriously inconsistent dependent type theory from 1971 [Martin-Löf(1971)]. Perhaps more significantly, it introduces a methodology for constructing and validating bidirectional type systems, illustrated with a nontrivial running example. Crucially, the fact that the system is not strongly normalizing is exploited to demonstrate concretely that the methodology relies in no way on strong normalization, which is perhaps peculiar given that bidirectional type systems are often (but not here) given only for terms in $\beta$-normal form [Pierce and Turner(2000)].

## 2 The 1971 Rules

Let us first see the system which we are about to reorganise.

**Really? Actually, I'm just guessing.**

| | | | |
|---|---|---|---|
| $f, s, t, S, T$ | $::=$ | $*$ | the type of all types |
| | $\mid$ | $(x : S) \to T[x]$ | dependent function spaces |
| | $\mid$ | $\lambda x : S.\ t[x]$ | typed abstraction |
| | $\mid$ | $f\ s$ | application |
| | $\mid$ | $x$ | variable*medskip* |
| $\Gamma, \Delta$ | $::=$ | $\mathcal{E}$ | empty context |
| | $\mid$ | $\Gamma, x : S$ | context extension, with freshly chosen $x$ |

It is my habit to be explicit (with square brackets) when introducing schematic variables in the scope of a binder: here, $T[x]$ and $t[x]$ may depend on the $x$ bound just before, whereas the domain type $S$ may not. It is, moreover, my habit to substitute such bound variables just by writing terms in the square brackets. For example, the $\beta$-contraction scheme is given thus:

$$(\lambda x : S.\ t[x])\ s \rightsquigarrow t[s]$$

The left-hand side is a *pattern*, which establishes schematic variables and makes clear their scope; the right-hand side is an *expression*, which must explain how the bound variable is instantiated.

Terms are identified up to $\alpha$-conversion and substitution is capture-avoiding: the formalization uses a scope-safe de Bruijn index representation [de Bruijn(1972)].

Let us define $\cong$, '$\beta$-convertability', to be the equivalence and contextual closure of $\rightsquigarrow$. The typing rules will identify types up to $\cong$.

We have two judgment forms

**context validity** $\boxed{\Gamma \vdash \textsc{ok}}$ asserts that $\Gamma$ is an assignment of types to distinct variables, where each type may depend on the variables given before;

**type synthesis** $\boxed{\Gamma \vdash t : T}$ asserts that the type $T$ can be *synthesized* for the term $t$.

$\boxed{\Gamma \vdash \text{OK}}$

$$\frac{}{\mathcal{E} \vdash \text{OK}} \qquad \frac{\Gamma \vdash \text{OK} \quad \Gamma \vdash S \ : \ *}{\Gamma, x : S \vdash \text{OK}}$$

$\boxed{\Gamma \vdash t \ : \ T}$

$$\frac{\Gamma, x : S, \Delta \vdash \text{OK}}{\Gamma, x : S, \Delta \vdash x \ : \ S} \qquad \frac{\Gamma \vdash \text{OK}}{\Gamma \vdash * \ : \ *}$$

$$\frac{\Gamma \vdash S \ : \ * \quad \Gamma, x : S \vdash T[x] \ : \ *}{\Gamma \vdash (x : S) \to T[x] \ : \ *} \qquad \frac{\Gamma \vdash S \ : \ * \quad \Gamma, x : S \vdash t[x] \ : \ T[x]}{\Gamma \vdash \lambda x : S.\ t[x] \ : \ (x : S) \to T[x]}$$

$$\frac{\Gamma \vdash f \ : \ (x : S) \to T[x] \quad \Gamma \vdash s \ : \ S}{\Gamma \vdash f \ s \ : \ T[s]}$$

$$\frac{\Gamma \vdash t \ : \ S \quad \Gamma \vdash T \ : \ * \quad S \cong T}{\Gamma \vdash t \ : \ T}$$

I do not write explicit variable freshness requirements. Rather, I think of the turnstile as equipped with a supply of fresh names for free variables, while bound names are arbitrary. So, for example, in the rule for typing an abstraction, it is not that we hope for a coincidence of bound names but that we impose a standard choice of a free name when we extend the context.

The system has one rule for each syntactic construct and one rule (the 'conversion' rule) to impose the identification of types up to convertability. If you look carefully at the rules for the syntax, you will see that the data left of the colon in the conclusion determine the data left of the colon in the premises; moreover, the data right of the colon in the premises determine the data right of the colon in the conclusion. That is to say that these five rules can be read as instructions for type synthesis. Only the conversion rule comes with no clear syntactic guidance: the essence of writing a type synthesis *algorithm* is to fix a particular strategy for deploying the conversion rule, then proving that strategy complete.

It is worth noting that the application rule has *two* occurrences of $S$ right of the colon: implicitly, such a rule demands that two synthesized types agree precisely, but the conversion rule allows them to be brought into precise agreement by computation. Meanwhile, the conversion rule allows a type, once synthesized, to be modified by any amount of forward *or backward* computation. Backward creates an opportunity to introduce any old nonsense, as

$$(\lambda X : *.\ *)\ (*\ *\ *\ *) \rightsquigarrow *$$

To prevent infection with such nonsense, the conversion rule insists that we check we end up with a valid type. Now, as it happens, our reduction system is confluent and moreover, forward computation preserves type. As a result, if we know that $S \cong T$ are valid types, then they have a common reduct $R$: we can compute $S$ to $R$ and $T$ to $R$ without stepping outside the valid types at any point. Hence, the conversion rule's check that $T$ is a type is both necessary and sufficient.

A further point of note is that the type synthesis rules have no axioms. The *only* axiom is that the empty context is uncontroversially valid. The two 'base cases' of typing, for $*$ and for variables, have premises ensuring context validity. The following 'sanity clauses' are admissible:

$$\frac{\Gamma \vdash t \ : \ T}{\Gamma \vdash \text{OK}} \qquad \frac{\Gamma \vdash t \ : \ T}{\Gamma \vdash T \ : \ *}$$

You can see that both of the rules which extend the context directly check the validity condition for so doing: the type synthesis rule for abstraction makes crucial use of the type annotation in $\lambda x : S.\ t[x]$, without which it would be necessary to guess the type of $x$ from its uses. The type of the abstraction body, being generic in the argument, allows us to form the correct function type unproblematically. Meanwhile, to see why synthesized types are well formed (for application in particular), we need stability of typing under substitution, which is as much as to say that we can substitute a (suitably weakened) typing derivation for some $s : S$ in place of all uses of the variable rule which witness $x : S$. Stability of typing under substitution relies, of course, on stability of

computation under substitution. However, our computation rule never makes any requirements about the presence of free variables, matching only syntactic constructs which are preserved by substitution, so it would be quite a surprise if stability under substitution were to fail.

The rule for $*$, often called 'Type-in-Type', opens the door to paradox. Famously, Girard had shown that the Burali-Forti paradox could be embedded in System U, which has two impredicative layers. Martin-Löf's system offered arbitrary impredicativity, making it easy to embed System U. However, despite being inconsistent and non-normalizing, this theory does enjoy the basic type preservation and progress properties we expect of functional programming languages, and many of the type theories we have today are effectively refinements with sufficient paranoia to prevent loops.

**Stick in the Hurkens construction?**

# 3 Type-*has*-Type

The idea behind bidirectional type systems is to make use of the way we sometimes know type information in advance. If we start from a type, there may be fewer choices to determine an inhabiting term. The type represents a *requirement*, rather than a *measurement*. We work a little more precisely at managing the flow of type information, and we gain some convenience and cleanliness. I like to start by separating the syntactic categories into checkable *constructions* and synthesizable *eliminations*.

| | | | |
|---|---|---|---|
| $s, t, S, T$ | $::=$ | $*$ | the type of all types |
| | $\mid$ | $(x : S) \to T[x]$ | dependent function spaces |
| | $\mid$ | $\lambda x.\ t[x]$ | untyped abstraction |
| | $\mid$ | $\underline{e}$ | embedded elimination |
| $e, f$ | $::=$ | $x$ | variable |
| | $\mid$ | $f\ s$ | application |
| | $\mid$ | $\vdots$ | to be continued... |

I have omitted one elimination form by way of generating a little suspense: let us see how we get along without it. Type formation and value introduction syntax sits on the *construction* side; variable usage and application sit on the *elimination* side. Eliminations embed into constructions, with a relatively unobtrusive but nontrivial underline: in a real implementation, there is no need to spend characters on this feature, but when studying metatheory, it helps to see where it is used. The reverse embedding is *not* available, and we shall see why when we study the rules.

At this stage, however, it is worth noting the following:

- the syntax forbids the expression of $\beta$-redexes;
- every elimination has a variable at its head, with a spine of arguments;
- it is syntactically invalid to substitute a construction for a variable.

We have two judgment forms:

**type checking** $\Gamma \vdash T \ni t$ constructions are checked with respect to a given type

**type synthesis** $\Gamma \vdash e \in S$ eliminations have their types synthesized, from the type of their head variable, which is given in the context

The 'forward' $\in$ of type synthesis is pronounced "in", with LaTeX macro `\in`; its reverse, used for checking, may be pronounced "ni", for its LaTeX macro is `\ni`, but might be more intelligibly pronounced "has" or "accepts".

Many other authors keep terms to the left of types and use arrows (directions vary) to make the checking/synthesis distinction. I insist on retaining the left-to-right flow of *time* through the syntax of judgments, which means that when checking, the type must come before the term.

In fact, I classify the schematic positions in judgment forms as having one of three *modes*:

**inputs** are given in advance and *required* to be valid (in some sense which should be specified);

**subjects** are the things under scrutiny, whose validity is the question;

**outputs** are data synthesized in the validation process and *guaranteed* to be valid (in some sense which should be clearly specified).

For the above judgment forms, we shall have

$$\underset{\text{input}}{\Gamma} \quad \vdash \quad \underset{\text{input}}{T} \quad \ni \quad \underset{\text{subject}}{t} \qquad\qquad \underset{\text{input}}{\Gamma} \quad \vdash \quad \underset{\text{subject}}{e} \quad \in \quad \underset{\text{output}}{S}$$

In order to specify the requirements and guarantees (but not to give the rules themselves), we shall also need a context validity judgment, $\boxed{\Gamma \vdash \textsc{ok}}$, for which $\Gamma$ is considered the subject. We should expect every judgment input to have a requirement for which it is the subject and every judgment output to have a guarantee for which it is the subject. Here,

$$\begin{array}{lrl} \Gamma \vdash T \ni t & \text{requires} & \Gamma \vdash \textsc{ok} \\ & \text{requires} & \Gamma \vdash * \ni T \\ \Gamma \vdash e \in S & \text{requires} & \Gamma \vdash \textsc{ok} \\ & \text{guarantees} & \Gamma \vdash * \ni S \end{array}$$

and we are correspondingly not free to write down any old rubbish by way of typing rules. Each rule gives rise to proof obligations which we must check. However, in the rule to establish a particular judgment, the requirements even to propose the judgment are *presumed*, not revalidated: as it were, "We would not be asking this question if we did not already know so-and-so.".

There is more to say about the impact of *mode* on valid notions of typing rule. Firstly, the inputs of conclusions and the outputs of premises must be *patterns*, which may match against any construct of the calculus *except free variables* and are the binding sites for the schematic variables of the rules. Secondly, the outputs of conclusions and the inputs of premises must be *expressions*, making use of the schematic variables in scope and instantiating any bound variables they may have. Scope flows clockwise round the rules, starting from the inputs of the conclusion, accumulating left-to-right through the premises, finishing with the outputs of the conclusion. Thirdly, only the schematic variables in the conclusion's *subjects* are in scope for the subjects of the premises, and they must all occur in at least one premise. Fourthly, a schematic subject variable becomes in scope for *expressions* only after it has been the subject of a premise (and thus achieved some measure of trust). These four conditions form the basis of a kind of 'religion' of typing rules: let us obey them for now and consider breaking them when we are older and more aware of the consequences of our actions.

The type checking and context validity rules are as follows:

$\boxed{\Gamma \vdash T \ni t}$

$$\frac{}{\Gamma \vdash * \ni *} \qquad \frac{\Gamma \vdash * \ni S \quad \Gamma, x{:}S \vdash * \ni T[x]}{\Gamma \vdash * \ni (x{:}S) \to T[x]} \qquad \frac{\Gamma, x{:}S \vdash T[x] \ni t[x]}{\Gamma \vdash (x{:}S) \to T[x] \ni \lambda x.\ t[x]}$$

$$\frac{\Gamma \vdash e \in S \quad S \equiv T}{\Gamma \vdash T \ni \underline{e}}$$

$\boxed{\Gamma \vdash \textsc{ok}}$

$$\frac{}{\mathcal{E} \vdash \textsc{ok}} \qquad \frac{\Gamma \vdash \textsc{ok} \quad \Gamma \vdash * \ni S}{\Gamma, x{:}S \vdash \textsc{ok}}$$

More rules will follow. For now, we start with 'Type-has-Type', without revalidating the context (for we *presume* its validity, given that the context is an input to the conclusion). Note that it is not only the case that our entitlements *allow* us to omit a $\Gamma \vdash \textsc{ok}$ premise, but also the case that our religion *forbids* such a premise, for it would have $\Gamma$ as its subject, and $\Gamma$ is an *input* variable, not in scope for the subjects of premises. In time, this will save our bacon.

Meanwhile, to check a function type, we check its components: once the domain has been checked, we may extend the context (maintaining its validity) and check the range. To check an

abstraction, we presume that the function type is indeed a type, and hence by inversion that its domain and range are types in the relevant contexts: we may thus proceed directly to checking that the range type accepts the body of the untyped abstraction, using the input domain as the type of the bound variable. Finally, to *check* an embedded elimination, we first *synthesize* the type of the elimination, and then insist that the two candidate types match *exactly* (i.e., that they are $\alpha$-equivalent, which is just syntactic identity for a de Bruijn representation), rather than up to some (so far unspecified, in any case) notion of conversion.

Now, some of you may wonder why we do not synthesize the types for $*$ and $(x : S) \to T[x]$, given that they clearly inhabit $*$ if they inhabit anything at all. While that works for this system with one universe, it is unstable with respect to overloading: type checking allows us to overload introduction forms for distinct types, including the overloading of type formation constructs within different universes. Such overloading rules out type synthesis but has no impact on type checking. The fix to restore normalization exactly requires us to introduce multiple universes and overload the function type constructor, so let us stick with type checking for these things.

Meanwhile, the heart of type synthesis is the variable rule, extracting the type of the head of an elimination from the context.

$$\frac{}{\Gamma, x : S, \Delta \vdash x \in S}$$

The synthesized type comes directly from the input context which is presumed valid, and must thus confirm that $S$ is indeed a type, which is what we must guarantee.

For application, we can get most of the way round before we run into trouble:

$$\frac{\Gamma \vdash f \in (x : S) \to T[x] \quad \Gamma \vdash S \ni s}{\Gamma \vdash f\ s \in ?}$$

Synthesizing the type (and we are guaranteed that it is a type) of the function, we may extract the (by inversion also a type) domain and use it to check the argument. However, when we come to give the output type of the whole application, the wheel comes off. We may be sure that $\Gamma, x : S \vdash * \ni T[x]$, and we surely want to give a type by choosing a suitable replacement for that $x$, but we may not give $T[s]$, as $s$ is not in the same syntactic category as $x$. Indeed, substituting such an $s$ for $x$ might create $\beta$-redexes which we have thus far excluded.

One possibility is to seek a derived notion of *hereditary* substitution [Watkins et al.(2003)Watkins, Cervesato, Pfenning which computes out the any such $\beta$-redexes on the fly, restoring syntactic legitimacy. However, we know that any such operation will not be well defined, as it can be persuaded to require the normalization of anything, and this language is surely non-normalizing. We might deal with the partiality of hereditary substitution by defining it *relationally*, but that is only to postpone the problem: in a non-normalizing calculus, hereditary substitution is not stable under hereditary substitution, as we can substitute an inert variable with just the term required to kick off an infinite computation. Correspondingly, the key stability property that drives the proof of type preservation will fail.

We had better think it out again.

# References


[de Bruijn(1972)] Nicolas G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.

[Martin-Löf(1971)] Per Martin-Löf. A theory of types. *Unpublished manuscript*, 1971.

[Pierce and Turner(2000)] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[Watkins et al.(2003)Watkins, Cervesato, Pfenning, and Walker] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The

propositional fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2003.