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 β-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.

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

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 β-contraction scheme is given thus:

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

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 α-conversion and substitution is capture-avoiding: the formalization uses a scope-safe de Bruijn index representation [de Bruijn(1972)].

Let us define $\cong$, ‘β-convertability’, to be the equivalence and contextual closure of $\leadsto$. The typing rules will identify types up to $\cong$.

We have two judgment forms

context validity $\boxed{\Gamma \vdash \text{OK}}$ asserts that Γ 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 variables 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. 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.

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.