Input ⟨ Subject ⟩ Output

ANONYMOUS AUTHOR(S)

ACM Reference Format:
Anonymous Author(s). 2025. Input ⟨ Subject ⟩ Output. 1, 1 (June 2025), 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This paper introduces a new standard by which to validate type systems, judging their judgments and ruling their rules. It is a discipline I have found helpful over the years, and it is long past time it was written down, so that others might find it helpful, too. Of course, it may not be to everyone's taste or serve everyone's purpose, but it should, at least, prove food for thought. In fact, I should like it to prove rather more. My purpose here is not to offer discipline for discipline's sake, but rather grow the class of metatheoretical properties attainable simply by conspicuously not doing anything to mess them up. This paper is thus a foray into metametatheory.

Let me give an example: stability under substitution. Substitution acts by replacing free variables with other things. If one is careful never to talk about free variables, one cannot accidentally say anything true which is falsifiable by the action of substitution. The common practice of writing lots of contexts Γ in typing rules is thus a risky practice, because it amounts to 'talking about free variables'. Local context *extension* with hypothetical judgments to go under a variable binding is perfectly safe, exactly because the variable binding gives protection from substitution. Stability under substitution is then nothing other than the lifting to typing derivations of a substitution action on terms, replacing appeals to hypothetical judgments about variables by actual derivations about the terms which replace the variables.

However, the key contribution of this paper is its analysis of the flow of information in two separable forms: *syntax* and *trust*. We may consider the instances of typing rules which make up a typing derivation to be a hierarchy of communicating actors, in accordance with

Dogma 1.1 (Client-Server). *A typing rule is a server for its conclusion and a client for its premises.*

The schematic variables we see in typing rules may thus be seen as standing for syntactic *signals* which are *directed*, either client-to-server or server-to-client. However, there is a second kind of communication at work, trading in *promises*. Every signal has not only a *sender*, but also a *guarantor* who makes a promise about it. We may thus distinguish the three[1] *modes* of the paper's title:

variety	sender	guarantor
input	client	client
subject	client	server
output	server	server

[1]A signal sent by a server in the hope of a promise from the client could make sense in an interactive setting and should be called an **abject**, as it amounts to a cry for help.

ACM XXXX-XXXX/2025/6-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 JUDGING JUDGMENTS

The flow of trust does not always align with the flow of syntax. We should be at pains to distinguish 'I am sending you a term about which *I* make a promise.' from 'I am sending you a term about which I hope *you* will make a promise.'. It is important that both of these things happen, and that we tell them apart. Consider, for example, the type theoretic judgment sans pareil:

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

It has two pieces of punctuation separating three *places*. Can we assign modes to the places? This question has multiple right answers, and which of them we choose has profound implications for the design of the syntax (where are type annotations necessary?) and for our requirements of the rules (what do they trust? what must they check?). For most purposes, t should be a subject[2], but what about Γ and T?

It is not unusual for presentations of type theories to treat all three places as subjects, and thus potential sources of misinformation. In this mode, we are required to ensure the admissibility of

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash} \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash T : \star}$$

where $\boxed{\Gamma \vdash}$ judges context validity (with Γ as subject) and $\star$ is the type of types. This choice results in rule systems where the only axiom is that the empty context is valid, while every atomic term is checked by a rule with a premise which validates the context. To take an example from Martin-Löf's 1971 (famously inconsistent) type theory:

$$\frac{}{\vdash} \qquad \frac{\Gamma \vdash S : \star}{\Gamma, x{:}S \vdash} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \star : \star}$$

The axiom is indeed the validity of the empty context, but the validity rule for context extension has, suspiciously, no premise $\Gamma \vdash$, as the premise which is present validates *both* Γ and S. Meanwhile the 'type-in-type' rule maintains the admissibility of context validity by brute force, and the admissibility of types being types by self-certification, like that bit in St Paul's epistle to Timothy which asserts that the Bible is all true.

If, instead, one takes Γ as an *input* to the typing judgment, we may spare ourselves the burden of revalidating it at every atom by daring to have trust. Of course, we should say *what* we trust.

Dogma 2.1 (Precondition). *Every input place in a judgment form must have an associated precondition judgment of which it is the subject.*

If we presume that it is the *client* for $\Gamma \vdash t : T$ who must ensure $\Gamma \vdash$, the above rules become

$$\frac{}{\vdash} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash S : \star}{\Gamma, x{:}S \vdash} \qquad \frac{}{\Gamma \vdash \star : \star}$$

The 'type-in-type' rule becomes an axiom, as Γ is presupposed to be valid. Meanwhile the context validity rule acquires the extra premise $\Gamma \vdash$, as that is no longer an admissible consequence of validating S — indeed, we have no *right* to validate S in Γ until we have validated Γ itself. We are entitled to assume the preconditions of a rule's conclusion, but we must check that we can meet the preconditions of premises. In the context extension rule, Γ and S come from the subject of the conclusion and are each the subject of a premise. Γ is validated as a subject *before* it is deployed as the trusted input of a premise. There is a discipline at work. Let us dig it out!

For a start, it is clear that we should also account for the properties of outputs.

[2]It is fun to consider t an output in the context of proof search or program synthesis.

Dogma 2.2 (Postcondition). *Every output place in a judgment form must have an associated postcondition judgment of which it is the subject.*

If we were to decide that the T in $\Gamma \vdash t : T$ were an output, we should certainly promise that $\Gamma \vdash T : \star$.

A judgment form should thus be specified as a Hoare triple, comprising

(1) what you should ensure before asking the question
(2) the question, with subjects clearly delimited in angle brackets, between inputs and outputs
(3) of what you are assured by the answer to the question

$$\boxed{\{\Gamma \vdash\} \quad \Gamma \vdash \langle\, t \,\rangle : T \quad \{\Gamma \vdash T : \star\}}$$

In other words, input ⟨ subject ⟩ output.

3 SUBJECT OR CITIZEN?

Trustworthy signals, whose sender and guarantor coincide, I refer to as *citizens*, whether they are inputs or outputs, by contrast with subjects. In many cases, the point of a judgment is to establish the citizenship of its subject, as we saw in our revised context extension rule, just now.

Subjects are purely syntactic entities, but citizens can be trusted to have a *semantics*. In particular, in a dependently typed setting, it is only the citizens which may compute. It makes sense to ask syntactic questions of subjects, e.g., 'Are you a function application?', but it is inappropriate to ask such questions of a citizen, because the property of being an application is readily destroyed by computation. Not only do citizens have a semantics: our treatment of them must respect their semantics.

Dogma 3.1 (Subjects versus Citizens). *A subject says; a citizen means. They stand to each other as pawns and queens.*

A subject can become a citizen once validated by a judgment, but there is no means of derogation from citizen to subject.

Dogma 3.2 (Trust for Citizens). *No premise of a typing rule may use a citizen in a subject position.*

That is to say, when we make Γ an input to the typing judgment, we not only make the context validation premises of rules for atoms unnecessary, we make them unacceptable. Taking this idea to its logical conclusion, we arrive at a particularly strong discipline.

Dogma 3.3 (Subject Linearity). *The subjects of a rule's premises must use all and only of the components of its conclusion's subjects exactly once.*

In effect, then we should not think of a subject as being a special kind of input. Rather it is *both* an input and an output. The premises of a rule send their subjects to be validated and receive the citizen counterparts of those subjects in return, for which we deftly use the same schematic variable names.

We can see subjects becoming citzens in more of the 1971 rules.

$$\frac{\Gamma \vdash S : \star \quad \Gamma, x{:}S \vdash T : \star}{\Gamma \vdash \Pi x{:}S.\, T : \star} \qquad \frac{\Gamma \vdash S : \star \quad \Gamma, x{:}S \vdash t : T}{\Gamma \vdash \lambda x{:}S.\, t : \Pi x{:}S.\, T}$$

In these rules, respectively for function type formation and function value construction, the function's domain, S is checked to be a type as the subject of the first premise, then used as a citizen to make a valid context extension in the second. In the rule for functions, S is also used as a citizen in the output of the conclusion, and we should check that these outputs really are types. For the

function type rule, we know that type is a type. For the fuction value rule, the first premise and the postcondition of the second are enough to ensure that we have a valid function type, by the function type rule.

Meanwhile, for function application, we have

$$\frac{\Gamma \vdash f : \Pi x{:}S_0.\, T \quad \Gamma \vdash s : S_1 \quad S_0 \equiv S_1}{\Gamma \vdash f\ s : T\lfloor s/x \rfloor}$$

where I am careful to note that the function domain and the argument type emerge from two distinct outputs and must be *checked* to be equal. There are plenty of choices about how to present that check, including by means of a judgment, but here by $\equiv$ I just mean syntactic equality up to renaming of bound variables (α-equivalence). We also see that the argument s is the subject of the second premise, but then substituted for x in the citizen output type. How do we show the postcondition of the conclusion? The postcondition of the first premise promises us a valid function type which, by inversion, must have a valid codomain T dependending on x. Our argument s has been checked to have the same properties as the hypothetical x, so if our system is stable under substitution, this instance of T remains a type.

Of course, we have not yet proven stability under substitution, and in due course, I shall show you how to avoid doing so. But we are not quite ready. Two rules are still missing from our example.

Firstly, we do not yet have any account of computation in types. Indeed, in the application rule, above, it may take computation to persuade S_0 and S_1 to coincide. This is often achieved by a 'conversion' rule, allowing arbitrary forward and backward computation:

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash S \cong T \quad \Gamma \vdash T : \star}{\Gamma \vdash s : T}$$

where $\cong$ is the congruence and equivalence closure of β-reduction. Allowing any old backward step risks introducing nonsense, which is why T has to be revalidated. I find both backwards computation and revalidation distasteful. In this formulation, T comes from nowhere and bears no promises. In our actual use case, we start from S_0 and S_1, both citizens known to be types.

Dogma 3.4 (Citizens Compute Forward). *Trust citizens to compute forwards. Never compute backwards.*

Let us do exactly that!

$$\frac{\Gamma \vdash s : S \quad S \triangleright S'}{\Gamma \vdash s : S'}$$

That is, we allow forward postcomputation for outputs, and we shall have to justify this rule by type soundness. Here, $\triangleright$ is the notion of *parallel reduction* derived systematically from the β-contraction scheme

$$(\lambda x{:}S.\, t)\ s \ \rightsquigarrow\ t\lfloor s/x \rfloor$$

Parallel reduction allows you to contract simultaneously none, some or all of the redexes you can currently see, but nothing further. I use it because it is pleasingly compositional — when a term parallel-reduces, so do all its subterms. So now, when we use the application rule in a derivation, we can use postcomputation in the function premise to obtain a function type in the first place, and then to compute its domain to something agreeable; we also use postcomputation in the argument premise to obtain a type which agrees!

The other missing rule is the rule for looking up variables in the context, which often looks like

$$\frac{}{\Gamma, x{:}S, \Gamma' \vdash x : S}$$

possibly with a context revalidation premise. I want this rule to stay missing! In the logical frameworks tradition of *hypothetical judgments*,I regard the context not as a lookup table, but as a collection of axioms locally in force. The context *lists* the variable *rules*. I shall have more to say about having less to say about the context in just a moment.

4 GET Γ GONE