Inputs ⟨ Subjects ⟩ Outputs

ANONYMOUS AUTHOR(S)

ACM Reference format:
Anonymous Author(s). 2022. Inputs ⟨ Subjects ⟩ Outputs. 1, 1, Article 1 (August 2022), 11 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 AN EXAMPLE FROM 1971

Let me give you a flavour of what is to come by reconstructing Martin-Löf's 1971 type theory, in a bidirectional style. I chose this system because it small enough to learn from, but also because it is inconsistent, ruling out reliance on totality of normalisation.

1.1 Syntax

Let us establish our grammar of terms. I define two sorts of *scoped* terms by mutual induction — a scope is a sequence of sorted variables which grows on the right.

$$
\begin{array}{rcl}
\textsc{chk}(\gamma) & ::= & \underline{\textsc{syn}(\gamma)} \\
& | & \star \\
& | & \Pi\ \textsc{chk}(\gamma)\ x.\textsc{chk}(\gamma, \textsc{syn}\ x) \\
& | & \lambda\ x.\textsc{chk}(\gamma, \textsc{syn}\ x) \\
\textsc{syn}(\gamma) & ::= & \textsc{chk}(\gamma)\ :\ \textsc{chk}(\gamma) \\
& | & \textsc{syn}(\gamma)\ \textsc{chk}(\gamma)
\end{array}
$$

I have specified this syntax in an informal 'scoped Backus-Naur form' notation. Each sort yields a nonterminal symbol carrying a scope, e.g. $\textsc{chk}(\gamma)$. I write ε for the empty scope. The notation $x.$ indicates a variable binding, where we name the variable in the grammar so that we can bring it into scope. As you can see, $\textsc{chk}(\gamma)$ embeds $\textsc{syn}(\gamma)$ and adds $\star$ (the type of types, dependent function types and Curry-style functions. Meanwhile, $\textsc{syn}(\gamma)$ includes type annotated terms in $\textsc{chk}(\gamma)$, and applications where the function is also in $\textsc{syn}(\gamma)$.

Implicitly, the grammar for every sort extends to include all the variables in scope of that sort. In particular $x \in \textsc{syn}(\gamma, \textsc{syn}\ x, \gamma')$. We can always α-convert to distinguish the variable names in scopes. A variable name is but a human-friendly way to indicate a position in a scope: machines fare better without them.

Scoped syntax is functorial with respect to the category of *thinnings* — order-preserving embeddings on scopes — exactly because scope lookup is covariant with respect to thinnings and scope extension is functorial on thinnings. I write $\theta : \gamma \leq \delta$ to indicate that θ witnesses such an

XXXX-XXXX/2022/8-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

embedding from γ to δ. You can think of θ as a vector of bits whose length is that of δ and whose 1s show which of δ's entries were embedded from γ.

Meanwhile, we acquire a category whose objects are scopes and whose morphisms $\gamma \Rightarrow \delta$ are simultaneous *substitutions* mapping every $\textsc{sort}\ x \in \gamma$ to some term in $\textsc{sort}(\delta)$. Again, scope extension acts functorially on substitutions, exactly because thinnings act on terms.

1.2 Typing Version α

For each of our syntactic sorts, let us have a judgment form whose purpose is to establish trust in terms of that sort. The grammar of *formal judgments* is

$$
\begin{array}{rcll}
\textsc{jud}(\gamma) & ::= & \textsc{chk}(\gamma) \ni \textsc{chk}(\gamma) & \text{checking a prior type} \\
 & | & \textsc{syn}(\gamma) \in \textsc{chk}(\gamma) & \text{synthesizing a posterior type} \\
 & | & \textsc{chk}(\gamma) = \textsc{chk}(\gamma) & \text{type equality} \\
 & | & x \in \textsc{chk}(\gamma) \vdash \textsc{jud}(\gamma, \textsc{syn}\ x) & \text{local extension}
\end{array}
$$

where entire *contexts* do not appear in formal judgments, only local extensions of contexts, recording what is known about a variable when we move under its binder. Note that thinnings act perfectly well on judgments, so we can bring everything we already knew into the current scope whenever we make a local extension. A local extension thus acts as an additional axiom, extending the rules just as bound variables extend terms, obviating the need to write a 'variable rule' at all. What are the rules for the other term forms? For checking, we have

$$
\frac{e \in S \quad S = T}{T \ni \underline{e}}
\qquad
\frac{}{\star \ni \star}
\qquad
\frac{\star \ni S \quad x \in S \vdash \star \ni T}{\star \ni \Pi\, S\, x.T}
\qquad
\frac{x \in S \vdash T \ni t}{\Pi\, S\, x.T \ni \lambda\, x.t}
$$

Meanwhile, for synthesis, we have

$$
\frac{\star \ni T \quad T \ni t}{t : T \in T}
\qquad
\frac{f \in \Pi\, S\, x.T \quad S \ni s}{f\ s \in \{x \mapsto s : S\}T}
$$

and, for the time being, we may take the equality judgment to be up to renaming of bound variables:

$$
\overline{T = T}
$$

Now, these rules are syntax directed in the t of $T \ni t$ and the e of $e \in S$. They have no extraneous premises checking, e.g., that the domain of an abstraction is a type. How are we to understand why these rules are sensible? Each judgment form has a *precondition* and a *postcondition*.

$\textsc{pre}(J)$	J	$\textsc{post}(J)$
$\star \ni T$	$T \ni t$	$\top$
$\top$	$e \in S$	$\star \ni S$
$\star \ni S \wedge \star \ni T$	$S = T$	$\top$
$\star \ni S \wedge x \in S \vdash \textsc{pre}(J)$	$x \in S \vdash J$	$x \in S \vdash \textsc{post}(J)$

Now, suppose we are working in a context of hypothetical judgments for each of which the precondition implies the postcondition (here, effectively that the types of the known variables are well formed). For any derivation of judgment J in such a context, we should like to know that $\textsc{pre}(J)$ is enough to ensure the pre- and postconditions for every judgment in the whole derivation. In other words, if we start well, we stay well. This property will hold for actual derivations of actual judgments if we can establish a suitable property of the rules from which derivations are composed. Let us find our way to that property.

To establish a judgment as the conclusion of a rule, one must demand that the premises hold. To demand a premise one must guarantee its precondition, but once the premise has been derived, one may rely on it and on its postcondition. Each judgment thus gives us an operator on propositions

$$\check{J}\;P = \textsc{pre}(J) \wedge ((J \wedge \textsc{post}(J)) \Rightarrow P)$$

which explains what use it is to demand J in the cause of establishing P. Of course, if one has a sequence of premises, $J_1\ \ldots\ J_n$, one can form the composition

$$(\circ_i \check{J}_i)\;P = \check{J}_1\,(\ldots(\check{J}_n\;P)\ldots)$$

which amounts to the assertion that the precondition for each premisefollows from the postconditions of the prior premises, and that P follows from all of their postconditions put together. Now, for a rule

$$\frac{J_1\quad\ldots\quad J_n}{J}$$

we must show why

$$\textsc{pre}(J)\ \Rightarrow\ (\circ_i \check{J}_i)\;(\textsc{post}(J))$$

i.e., that if we assume it is reasonable to enquire after the rule's conclusion in the first place, then it is also reasonable to enquire after the premises and, upon their successful derivation, to deliver the conclusion's postcondition. For an axiom, with zero premises, this reduces to $\textsc{pre}(J) \Rightarrow \textsc{post}(J)$, so our assumption about the context amounts to treating hypothetical judgments as local axioms.

Let us visit each of our rules in turn.

- $\dfrac{\underline{e} \in \underline{S}\quad \underline{S = T}}{T \ni \underline{e}}$ We are given $\star \ni T$. The first premise has nothing to check, and it gives us $\star \ni S$. So we have established the precondition for checking $S = T$.
- $\dfrac{}{\star \ni \star}$ The postcondition for this rule is derived by this rule!
- $\dfrac{\star \ni S\quad x \in S \vdash \star \ni T}{\star \ni \Pi\, S\; x.T}$ We are given $\star \ni \star$, which we know anyway, and that is what we must deliver to invoke the first premise. Once we know $\star \ni S$, we may extend the context with $x \in S$. Again, $\star \ni \star$ for the second premise.
- $\dfrac{x \in S \vdash T \ni t}{\Pi\, S\; x.T \ni \lambda\; x.t}$ We know $\star \ni \Pi\; S\; x.T$, so by inversion, $\star \ni S$ and $x \in S \vdash \star \ni T$. Correspondingly, we may extend the context with $x \in S$ and then check $T \ni t$.
- $\dfrac{\star \ni T\quad T \ni t}{t\ :\ T \in T}$ For the first premise, we need $\star \ni \star$. For the second premise we need the first premise. The conclusion postcondition is again the first premise.
- $\dfrac{f \in \Pi\, S\; x.T\quad S \ni s}{f\;s \in \{x \mapsto s\ :\ S\}T}$ The first premise promises us $\star \ni \Pi\; S\; x.T$, so by inversion, $\star \ni S$ and $x \in S \vdash \star \ni T$. We may thus invoke the second premise. Now, for the conclusion postcondition, we may deduce $s\ :\ S \in S$ from $\star \ni S$ and $S \ni s$. Substituting this derivation for all uses of the hypothetical $x \in S$, we obtain $\star \ni \{x \mapsto s\ :\ S\}T$.

In that last case, I made use of the fact that substitution extends from terms to derivations. How do I know? I have been very careful to ensure that none of my rules ever says anything about *free* variables. Substitutions preserve everything but free variables, so the only parts of derivations where they have noticeable impact are where we appeal to hypothetical judgments. Stability under substitution is exactly that if a substitution preserves all the hypothetical judgments, then it preserves all derivations. Note that an ill typed substitution may falsify the preconditions for a derivation to be meaningful, but it will still preserve the derivation.

So, we get out as much sense as we put in! However, this system does no computation, so we had better add it and see what sort of mess it makes.

1.3 Computation

Let us define some *contraction schemes*, from which we shall extract a notion of untyped conversion. Of course, there are more sophisticated ways to account for computation, but let us begin with the basics.

$$(\upsilon) \quad \underline{t : T} \rightsquigarrow t \qquad\qquad (\beta) \quad (\lambda\, x.t : \Pi\, S\, x.T)\, s \rightsquigarrow \{x \mapsto s : S\}(t : T)$$

The idea here is that function types drive function applications. The υ rule deletes the type annotation from a term which is not being applied. The β rule uses the type information in the redex to annotate the reduct and also to annotate the argument so that, wherever the variable is substituted, further computation may be enabled.

We might think of a contraction scheme as saying that any substitution instance of the left-hand side contracts to the corresponding substitution instance of the right-hand side, but where do these substitutions come from? They come from *matchings*: a matching is a formula (such as we write in these rules) with each schematic variable annotated by a term that it maps to. E.g.,

$$\underline{\{t \mapsto \lambda\, x.\underline{x}\} : \{T \mapsto \Pi\, \star\, \_.\star\}}$$

is a matching for υ-contraction. Matchings support three erasures:

- if you replace $\{v \mapsto t\}$ by v, you get a formula which is the *pattern* of the matching;
- if you replace $\{v \mapsto t\}$ by t, you get a term which is the *instance* of the matching;
- if you erase everything outside the $\{v \mapsto t\}$s, you get the *substitution* of the matching.

Moreover, we call any subterm of any t in a matching annotation $\{v \mapsto t\}$ a *residual* of the matching, and we can be sure that such residuals are copied intact into the matching substitution. So, a *redex* is a matching instance of the left-hand side of a contraction scheme. We say the contraction schemes are *orthogonal* if no term is a redex for two of them, and whenever a redex has a proper subterm which is also a redex, the latter is a residual of the former's matching. E.g., υ and β are clearly orthogonal, because embedding, $\underline{-}$, occurs only and outermost in υ while application, $-\ -$, occurs only and outermost in β.

Orthogonal contraction schemes give rise to confluent reduction systems, essentially by Takahashi's proof. Once we have contraction schemes, we can systematically derive an inductive definition of *parallel reduction*. We begin with structural rules for variables and all syntactic productions, thus ensuring that the relation is at least reflexive and closed under all syntactic contexts.

$$\frac{}{x \rhd x}$$

$$\frac{e \rhd e'}{\underline{e} \rhd \underline{e'}} \qquad \frac{}{\star \rhd \star} \qquad \frac{S \rhd S' \quad T \rhd T'}{\Pi\, S\, x.T \rhd \Pi\, S'\, x.T'} \qquad \frac{t \rhd t'}{\lambda\, x.t \rhd \lambda\, x.t'}$$

$$\frac{t \rhd t' \quad T \rhd T'}{t : T \rhd t' : T'} \qquad \frac{f \rhd f' \quad s \rhd s'}{f\, s \rhd f'\, s'}$$

Then, we add rules generated from each contraction schemes by renaming all the schematic variables in the reduct and giving premises which allow the schematic variables in the premises to reduce to their renamed variants.

$$\frac{t \rhd t'}{\underline{t : T} \rhd t'} \qquad \frac{t \rhd t' \quad S \rhd S' \quad T \rhd T' \quad s \rhd s'}{(\lambda\, x.t : \Pi\, S\, x.t)\, s \rhd \{x \mapsto s' : S'\}(t' : T')}$$

We can also define the *development*, $\mathbf{dev}(t)$, of a term t, which aggressively contracts redexes from the outside in.

$$\begin{array}{lcl}
\mathbf{dev}(\underline{t\ :\ T}) & = & \mathbf{dev}(t) \\
\mathbf{dev}((\lambda\, x.t\ :\ \Pi\, S\ x.T)\ s) & = & \{x \mapsto \mathbf{dev}(s)\ :\ \mathbf{dev}(S)\}(\mathbf{dev}(t)\ :\ \mathbf{dev}(T)) \\
\multicolumn{3}{l}{\textit{and otherwise,}} \\
\mathbf{dev}(x) & = & x \\
\mathbf{dev}(\underline{e}) & = & \underline{\mathbf{dev}(e)} \\
\mathbf{dev}(\star) & = & \star \\
\mathbf{dev}(\Pi\, S\ x.T) & = & \Pi\, \mathbf{dev}(S)\ x.\mathbf{dev}(T) \\
\mathbf{dev}(\lambda\, x.t) & = & \lambda\, x.\mathbf{dev}(t) \\
\mathbf{dev}(t\ :\ T) & = & \mathbf{dev}(t)\ :\ \mathbf{dev}(T) \\
\mathbf{dev}(f\ s) & = & \mathbf{dev}(f)\ \mathbf{dev}(s)
\end{array}$$

By construction, $t \triangleright \mathbf{dev}(t)$. Moreover, whenever we develop a redex, we develop all of its residuals, too. By orthogonality, that means we develop all the redexes present in the input, and as a consequence, whenever $s \triangleright t$, we also have $t \triangleright \mathbf{dev}(s)$. We acquire the *diamond* property for $\triangleright$

$$\forall s, p, q.\ s \triangleright p \wedge s \triangleright q \implies \exists r.\ p \triangleright r \wedge q \triangleright r$$

by taking $r = \mathbf{dev}(s)$. This extends to the same for $\triangleright^*$, the reflexive-transitive closure of $\triangleright$, by tiling a parallelogram with diamonds. We further obtain that the equivalence closure of $\triangleright$ amounts to having a common reduct.

With this machinery in place, we may add computation to our theory in the form of two new rules. We may compute a type before checking it or after synthesizing it.

$$\frac{T \triangleright T' \quad T' \ni t}{T \ni t} \qquad \frac{e \in S \quad S \triangleright S'}{e \in S'}$$

Note that, with these rules in place, we may construct derivations with multiple computation steps on either side of the $S = T$ check, so that we effectively allow type conversion at the change of direction. Everywhere we insist on a particular type form, $\star$ or $\Pi\, S\ x.T$, we allow only reduction, but we also have that whenever a type is convertible to a canonical form, it can reach a convertible canonical form by reduction alone.

However, if we want to argue that preconditions ensure postconditions, we shall have to account for the impact of computation, which is our next task.

1.4 Subject Reduction

Judgments establish trust, so it would be unfortunate, to say the least, if computation were to damage that trust. Of course, we can speak only to forward computation: goodness knows monstrous arguments a backward β-step might conjure for a vacuous abstraction. It is fortunate, then, that our rules require only forward computation. The tricky part of proving that forward computation preserves judgments is the movement of subterms between the judgments' places: application copies the argument into the type, function type formation copies the domain into the context, and so on. If we are to proceed by mutual induction on derivations, we shall need to be sufficiently flexible in the statement of our goals, if our induction hypotheses are to be fit for purpose.

Like a rabbit from a hat, I produce the following claims:

- Suppose that under some $x_i : S_i$ we have $T \ni t$, and that $S_i \triangleright^* S_i'$, $T \triangleright^* T'$ and $t \triangleright t'$. Then under $x_i : S_i'$ we have $T' \ni t'$.
- Suppose that under some $x_i : S_i$ we have $e \in S$, and that $S_i \triangleright^* S_i'$, $e \triangleright e'$. Then there exists some S' such that $S \triangleright^* S'$ and under $x_i : S_i'$ we have $e' \in S'$.

That is, if we know a judgement and we allow the devil to compute contexts and checked types forwards arbitrarily, but the terms only by *one* parallel reduction step, then we may recover our judgment by sufficiently computing synthesized types. If we let the devil compute the context, we shall certainly need to control the computation of synthesized types, if we are to have any chance of catching up with appeals to hypothetical judgments: when the devil computes $S_i \rhd^* S'_i$, we must be able to deploy a copycat strategy to recover $x_i \in S_i$ as $x_i \in S'_i$. Let us see how the game plays out for the rest of the rules: we obtain a case for each term former taking a *structural* parallel reduction step, together with cases for actual contraction.

- $\dfrac{e \in S \quad S = T}{T \ni \underline{e}} \quad T \rhd^* T' \quad e \rhd e'$ We must show $T' \ni \underline{e'}$.

  By induction hypothesis, $e' \in S'$ for some $S \rhd^* S'$. As $S = T$, confluence gives us a common reduct, U for S' and T'. The precomputation rule allows us to derive $T' \ni \underline{e'}$ from $U \ni \underline{e'}$, which follows by change of direction from $e' \in U$, derived by postcomputation from $e' \in S'$.
- $\dfrac{}{\star \ni \star} \quad \star \rhd^* \star \quad \star \rhd \star$ We must show $\star \ni \star$, which holds.
- $\dfrac{\star \ni S \quad x \in S \vdash \star \ni T}{\star \ni \Pi\, S\, x.T} \quad \star \rhd^* \star \quad \Pi\, S\, x.T \rhd \Pi\, S'\, x.T'$ We must show $\star \ni \Pi\, S'\, x.T'$.
  We had $\star \ni S$ and $x \in S \vdash \star \ni T$ in the typing derivation, and $S \rhd S'$ and $T \rhd T'$ from reduction. One induction hypothesis yields $\star \ni S'$; the other yields $x \in S' \vdash \star \ni T'$ as we may compute in the context. We may thus use the same rule to deduce the goal.
- $\dfrac{x \in S \vdash T \ni t}{\Pi\, S\, x.T \ni \lambda\, x.t} \quad \Pi\, S\, x.T \rhd^* \Pi\, S'\, x.T' \quad \lambda\, x.t \rhd \lambda\, x.t'$ We must show $\Pi\, S'\, x.T' \ni \lambda\, x.t'$.
  Allowing for precomputation, we must have had $x \in S \vdash T'' \ni t$ for some $T \rhd^* T''$ from the typing derivation. From reduction, we must have $S \rhd^* S'$ and $T \rhd^* T'$. Taking T''' as the common reduct of T' and T'', we may derive $x \in S' \vdash T''' \ni t$ by induction hypothesis and obtain the goal by precomputation and abstraction.
- $\dfrac{\star \ni T \quad T \ni t}{t : T \in T} \quad t : T \rhd t' : T'$ We must show $t' : T' \in R$ for some $T \rhd^* R$.
  We must have had $t \rhd t'$ and $T \rhd T'$. By induction hypothesis, $\star \ni T'$ and, computing the type, $T' \ni t'$. We thus choose $R = T'$ and reapply the same rule.
- $\dfrac{f \in \Pi\, S\, x.T \quad S \ni s}{f\ s \in \{x \mapsto s\ :\ S\}T} \quad f\ s \rhd f'\ s'$ We must show $f'\ s' \in R$ for some $\{x \mapsto s\ :\ S\}T \rhd^* R$.
  By induction, we obtain $f \in \Pi\, S'\, x.T'$ for some $S \rhd^* S'$ and $T \rhd^* T'$. Computing the type, our other hypothesis yields $S' \ni s'$. We may thus take $R = \{x \mapsto s'\ :\ S'\}T'$ because computation is stable under substitution, and reapply.
- $\dfrac{T \rhd T' \quad T' \ni t}{T \ni t} \quad T \rhd^* T'' \quad t \rhd t'$ We must show $T'' \ni t'$.
  Confluence gives T''' as common reduct of T' and T'', so induction yield $T''' \ni t'$, then precomputation yields the goal.
- $\dfrac{e \in S \quad S \rhd S'}{e \in S'} \quad e \rhd e'$ We must show $e' \in R$ for some $S' \rhd^* R$.
  By induction, $e' \in S''$ for some $S \rhd^* S''$. Confluence yields S''' as common reduct of S' and S'''. We take $R = S'''$ and derive goal from hypothesis by postcomputation.
- $\dfrac{t : T_0 \in T_1 \quad T_1 = T}{T \ni \underline{t : T_0}} \quad T \rhd^* T' \quad \underline{t : T_0} \rhd t'$ where $t \rhd t'$ We must show $T' \ni t'$.

  We must have had $T_0 \ni t$ and $T_0 \rhd^* T$. Hence $T_0 \rhd^* T'$ and the induction hypothesis yields the goal.

- $\dfrac{(\lambda\, x.t \;:\; \Pi\, S'\, x.T') \in \Pi\, S\, x.T \quad S \ni s}{(\lambda\, x.t \;:\; \Pi\, S'\, x.T')\; s \in \{x \mapsto s \;:\; S\}T}$

  $(\lambda\, x.t \;:\; \Pi\, S'\, x.T')\; s \triangleright \{x \mapsto s' \;:\; S''\}(t' \;:\; T'')$ where $s \triangleright s' \quad S' \triangleright S'' \quad T' \triangleright T'' \quad t \triangleright T'$
  We must show $\{x \mapsto s' \;:\; S''\}(t' \;:\; T'') \in R$ for some $\{x \mapsto s \;:\; S\}T \triangleright^* R$.
  We must have had $S' \triangleright^* S$ and $T' \triangleright^* T$. Further, we must have had $x \in S_0 \vdash T_0 \ni t$ for some $S' \triangleright^* S_0$ and $T' \triangleright^* T_0$. Moreover, we must have had subderivations of $\star \ni S'$ and $x \in S' \vdash \star \ni T'$. By confluence, obtain common reducts S_c and T_c. By induction, obtain $S_c \ni s'$ and $x \in S_c \vdash T_c \ni t'$. Precomputation yields $S'' \ni S'$ and $x \in S_c \vdash T'' \ni t'$. Induction yields $\star \ni S''$ and $x \in S_c \vdash \star \ni T''$ as $S' \triangleright S''$ and $T' \triangleright T''$. We thus obtain $s' \;:\; S'' \in S_c$ and $x \in S_c \vdash t' \;:\; T'' \in T_c$. Stability under substitution tells us that $R = \{x \mapsto s' \;:\; S''\}T_c$ yields the goal. We could also choose $R = \{x \mapsto s' \;:\; S_c\}T_c$.

So, my lucky rabbit did the trick! Whenever I needed an induction hypothesis, the term in question had computed only by parallel reduction. Whenever checked types precomputed or synthesized types postcomputed in competing ways, confluence always gave me a common reduct which I could then choose either in a checking hypothesis or a synthesis conclusion. It is almost as if the places in the judgments have some sort of *orientation* which aligns with the rules for computation and the statements of subject reduction!

With subject reduction in place, it is straightforward to see that the preconditions and postconditions which demand validity of types are now stable with respect to the forward computations permitted by the rules. Repairing the naïve proof to cope with computation is safely left as an exercise.

Now, the real purpose of this paper is to demystify the alignment by which the above proof comes out. Let us look not at the rabbit but rather at the hat I pulled it from, for this trick is not magic, but millinery!

2 JUDGING JUDGMENTS; RULING RULES

Bidirectional type systems come with a sense that checked types flow 'inward' while synthesized types flow 'outward', but it is not at all standard to bake this distinction into the very idea of what it is to be a judgment form. In our statement of subject reduction, we quantified universally over how the inputs computed and existentially over how the outputs computed, so perhaps the distinction is worth observing. More subtly, we have some places in judgment forms to which we attach conditions explaining what we rely on or guarantee for the things in those places, but others with no such condition; and in our example, that distinction coincided with the places which had $\triangleright^*$ in the statement of subject reduction and those which were permitted only $\triangleright$. Let us refine the notion of *judgment* form to make this analysis more explicit, then explore the consequences of doing so.

2.1 Modes for Judgment Forms

We may think of a judgment as a structured interaction between a *client* who *proposes* and a server who *avers*. The data in its places are transmitted either from client to server or vice versa, but moreover, some sort of *guarantee* is made about the data by one to the other. We may classify judgment form places with a *mode* according to the senders of their data and the makers of the guarantees about them:

Mode	**Sender**	**Guarantor**
Input	client	client
Subject	client	server
Output	server	server

We might imagine a fourth ‘whataboutery’ mode in which the server sends data to the client in the hope of a guarantee, but it seems unlikely to play a role in typechecking interactions, at least for complete terms, so let us let slip that idea for the present. Likewise, one might imagine judgment forms with arbitrary interaction protocols, but we will get a long way with only those whose inputs precede their subjects which in turn precede the outputs.

Let us specify judgment forms uniformly for all scopes by writing grammar productions using syntactic sorts for nonterminal symbols, punctuation as we see fit, a ‘$\langle$’ between inputs and subjects as far right as possible, and a ‘$\rangle$’ between subjects and outputs as far left as possible after the ‘$\langle$’. Our example system has

$$\begin{gathered}\textsc{chk} \ni \langle\textsc{chk}\rangle\\ \langle\textsc{syn}\rangle \in \textsc{chk}\\ \textsc{chk} = \textsc{chk}\langle\rangle\end{gathered}$$

2.2 Contracts and Contexts

We may think of a judgment with subjects as a *validator* for those subjects. The guarantees made by clients about inputs and servers about outputs are validity guarantees. Each judgment form must thus have a *contract* of preconditions and postconditions: the preconditions are for each input a judgment in which that input stands as a subject; the postconditions are for each output a judgment in which that output stands as a subject.

We shall need at least those hypothetical judgments about a variable in which it stands as a subject, for how else are we to validate the uses of that variable? That is why, when we bind variables x in sort $\textsc{syn}$, we take hypotheses of form $x \in S$. We thus acquire that derivations respect well sorted simultaneous substitutions, provided their hypotheses similarly substituted are derivable: the latter derivations, suitably thinned, substitute into places where the hypothetical judgments were invoked, exactly extending substitutions from the syntax of terms to the derivations of judgments. It is also reasonable to consider hypothetical judgments in which the variable of concern stands as an input, e.g., equipping a variable already declared with a *definition*. What matters is to specify the allowable instantiations of the variable compatibly with this grafting of derivations.

Hypothetical judgments must keep their contracts, too, so it is a duty for any rule which has a premise in a locally extended context to ensure that the corresponding contract is met. In our example, this means that whenever we say $x \in S \vdash \ldots$ we must be able to explain why $\star \ni S$, because that is what the type synthesis postcondition promises.

2.3 Patterns versus Expressions

In inference rules as we know them, we write formulae which employ *schematic variables* to stand for the concrete objects for which the rules may be used to construct derivations. A rule is valid for any syntactically correct instantiation of its schematic variables. That is, the schematic variables are bound *implicitly* by universal quantifiers under which the rule amounts to the implication of its conclusion from the conjunction of its premises. When deploying rules to make derivations, we are obliged to instantiate those implicitly quantified variables with whatever insight and inspiration we can muster. For example, we might give a rule for constructing dependent pairs as follows:

$$\frac{s : S \quad t : \{x \mapsto s\}T}{s, t : \Sigma\, S\, x.T}$$

(danger!) Given s and t, one constructs derivations by dreaming up a suitable S and a suitable T depending on x, but how is this dreaming to be done? If we are *checking* types, it is reasonable to dismantle the pair type to extract S and $x.T$. However, if we are *synthesizing*, we may readily obtain S from the first premise, but the second premise gives us but one substitution instance of T, from which there is no general way to infer the dependency pattern. Inverting substitution is more magic than you have the right to expect!

Our modes make it clear who sends and who receives the data in a typing rule:

> *A rule is the server for its conclusion and the client for its premises.*

Hence, each rule receives the inputs and subjects of its conclusion, sends the inputs and subjects of its premises, receives the outputs of its premises, and sends the outputs of its conclusion. These communications can happen, in principle, in any causally sustainable order, but it is often an adequate simplification to demand that they happen *clockwise*, starting from wherever in the conclusion has the ⟩ in its judgment form.

We may now make a distinction between the *patterns* by which rules may *analyse* the data they receive and the *expressions* with which they *construct* the data they send. We receive, in return, two benefits.

Firstly, implicit universal quantification of the schematic variables gives way to the explicit binding of each schematic variables in exactly one pattern, making it clear who is respnsible for delivering that information, and just as clear what is *in scope* at the time. The use sites of schematic variables should be in expressions only, and then only in places where the variables in scope for them are either captured or substituted. Indeed, it is clear by inspection which variables are not in scope at use sites and must thus be substituted, hence we may write substitutions without explicit naming, e.g.

$$\frac{f \in \Pi\, S\, x.T \qquad S \ni s}{f\ s \in \{s\ :\ S\}T}$$

is perfectly meaningful, because the binding site of T is in the first premise with x in scope, but what is missing from scope in the conclusion's output is exactly such an x, so there is no doubt as to what is being substituted by $s\ :\ S$.

Secondly, we become free to gain reassurance at the expense of power by reducing the expressivity of the pattern language to avoid magic that we are better off without. At the very least, we should disallow general substitutions in patterns, exactly because we cannot invert them. In our application example, the substitution is safely in an expression position; in our dependent pair example, if we read : as ∈, we see that the second premise's substitution occurs in a pattern position and is thus unwise. Later, we should also consider how further restricting the pattern language in some places might assist in establishing metatheoretical properties, e.g., by ensuring that we never demand a type matches a pattern that computation can destroy. As in 'Theorems for Free', the less we can look, the more we are promised. When ignorance is bliss, 'tis folly to be wise.

2.4 Citizens or Subjects?

So far, it might seem that inputs and subjects differ only morally: who makes what guarantee? However, the distinction runs deeper. The schematic variables brought into scope by matching on a conclusion input or a premise output are *citizens*, in (meta-)scope for use in subsequent premise input or conclusion output expressions. By contrast, schematic variables from matching on a conclusion subject are not citizens, but merely subjects themselves, and they may not, ab initio, be used in premise inputs or conclusion outputs, because subjects are unvalidated.

What promotes a subject to a citizen is its validation, i.e., its use as the subject of a premise. Indeed, every premise subject must be a subject schematic variable, and every subject schematic variable must be a premise subject exactly once. This linear discipline requires us to design judgment forms which characterize what it is to validate a thing. Moreover, the stipulation that premise subjects come from conclusion subjects prevents *re*validation of data from inputs or outputs which should already be covered by a *contract*. There is no ‘papiere bitte’ for citizens!

This design choice marks a shift from standard practice in type theory. It is common to insist that $\Gamma \vdash t : T$ must imply Γ VALID and $\Gamma \vdash T$ TYPE, and thence to require additional premises e.g., revalidating the context for each atomic term. Whilst commendably diligent, this conventional choice has an unfortunate consequence for proofs by induction on derivations: it removes our ability to work with goals which give citizens more freedom than subjects. If a conclusion input can become a subject premise, then we are obliged to deduce an induction conclusion about a thing with a citizen’s freedom to misbehave than our subject induction hypothesis can address. Concretely, in our earlier proof of subject reduction, the subjects were allowed only $\rhd$, so that their subterms do only $\rhd$, but the citizens were permitted arbitrary $\rhd^*$ computation. By ensuring that our rules do not revalidate citizens, we avoided any risk of seeking a $\rhd^*$ conclusion from a mere $\rhd$ hypothesis. If you ask whether $\Gamma \vdash t : T$ without first ensuring that Γ VALID and $\Gamma \vdash T$ TYPE, you are the author of your own misfortune.

3 GENERIC METATHEORY

Virtue may be its own reward, but I like to get paid. Let us formulate a framework which rewards us with metatheorems if we play by its rules.

3.1 Terms and Expressions

Fix a set SORT of syntactic sorts, and another set ATOM of terminal symbols. Let us give a way to associate with each sort a description of a syntax with binding.

DEFINITION 3.1 (SYNTAX DESCRIPTIONS). *The set* DESC *of* ***syntax descriptions*** *with respect to* SORT *is given inductively as follows:*

$$\begin{array}{rcll} \textit{DESC} & \ni & 1 & \textit{for the empty tuple} \\ & | & \textit{DESC} \times \textit{DESC} & \textit{for pairing} \\ & | & \textit{SORT} & \textit{for a subterm of a given sort} \\ & | & \textit{SORT}\,.\ \textit{DESC} & \textit{for binding a variable of a given sort} \end{array}$$

A ***syntax*** *is then given by a function in* SORT $\longrightarrow$ LIST(ATOM $\times$ DESC)*, mapping each sort to a choice of atomic tags accompanied a description of their associated payload.*

EXAMPLE 3.2 (BIDIRECTIONAL 1971 MARTIN-LÖF TYPE THEORY). *Our example type theory takes* SORT $= \{$CHK, SYN$\}$*, and you will see some of the atoms, shortly. Its syntax is*

$$\begin{array}{rcll} \textit{CHK} & \mapsto & [\,(\texttt{type}, & 1) \\ & & ,(\texttt{pi}, & \textit{CHK} \times (\textit{SYN}\,.\ \textit{CHK})) \\ & & ,(\texttt{lambda}, & \textit{SYN}\,.\ \textit{CHK}) \\ & & ,(\texttt{embed}, & \textit{SYN}) \\ & &] & \\ \textit{SYN} & \mapsto & [\,(\texttt{check}, & \textit{CHK} \times \textit{CHK}) \\ & & ,(\texttt{apply}, & \textit{SYN} \times \textit{CHK}) \\ & &] & \end{array}$$

DEFINITION 3.3 (SCOPES AND SCOPED TERMS). *The category* ***Sco*** *of scopes has as objects* γ, δ : LIST(SORT) *with morphisms being thinnings,* $\theta, \phi : \gamma \leq \delta$*. Terms are given as a family of functors* $(\textbf{Sco} \longrightarrow \textbf{Set})^{\textit{SORT}}$*. Given such a family,* R*, we may interpret some* D : DESC *as a functor* $[\![D]\!](R, \cdot)$:

$\textbf{\textit{Sco}} \longrightarrow \textbf{\textit{Set}}$ *as follows:*

$$\begin{array}{lcl}
[\![1]\!](R,\gamma) & = & 1 \\
[\![D \times D']\!](R,\gamma) & = & [\![D]\!](R,\gamma) \times [\![D']\!](R,\gamma) \\
[\![s]\!](R,\gamma) & = & R(s,\gamma) \\
[\![s\,.\ D]\!](R,\gamma) & = & [\![D]\!](R,(\gamma,s))
\end{array}$$

Observe that the above is strictly positive in R. Hence, given a syntax F, we may construct its terms as a least fixpoint:

$$\begin{array}{rcl}
\textbf{\textit{Term}}(s,\gamma) & = & \{\#\theta \mid \theta \in [s] \leq \gamma\} \\
& \cup & \{(a,t) \mid (a,D) \in F(s), t \in [\![D]\!](\textbf{\textit{Term}},\gamma)\}
\end{array}$$

Note that a thinning $\phi : \gamma \leq \delta$ acts on a term t as $t\phi$, where $(\#\theta)\phi = \#(\theta;\phi)$. On other terms, ϕ acts structurally; to go under a binder, note that $\phi, 1 : \gamma, s \leq \delta, s$.

In human company, it helps to write $x.t$ for a term in $[\![.\ ]\!]D$, with x written in place of $\#\vec{0}, 1, \vec{0}$ for those variable use sites which refer to that binding, as long as the name x is carefully chosen to be unambiguous.

These terms constitute the *object* language that rules talk about, but they are not enough to give the *meta*-language that rules use to talk *about* terms, with schematic variables standing for terms unknown and substitutions instantiating object variables. Schematic, or ‘meta-’, variables should be scoped to indicate the object variables on which they may depend. Whenever a metavariable is used, its permitted dependencies should be substituted by expressions well scoped at the use site.

Definition 3.4 (Expressions and Substitutions). *Given some scope-indexed set $M(\cdot) : |\textbf{\textit{Sco}}| \longrightarrow \textbf{\textit{Set}}$, giving a notion of metavariables whose permitted dependencies are fixed, we acquire expressions and simultaneous substitutions, $\textbf{\textit{Expr}}_M : (\textbf{\textit{Sco}} \longrightarrow \textbf{\textit{Set}})^{\mathrm{SORT}}$ and $\cdot \Rightarrow \cdot : \textbf{\textit{Sco}}^{op} \times \textbf{\textit{Sco}} \longrightarrow \textbf{\textit{Set}}$, by mutual induction:*

$$\begin{array}{rcl}
\textbf{\textit{Expr}}_M(s,\gamma) & = & \{\#\theta \mid \theta \in [s] \leq \gamma\} \\
& \cup & \{(a,t) \mid (a,D) \in F(s), t \in [\![D]\!](\textbf{\textit{Expr}}_M,\gamma)\} \\
& \cup & \{m(\sigma) \mid \exists \xi.m \in M(\xi), \sigma \in \xi \Rightarrow_M \gamma\} \\
\varepsilon \Rightarrow_M \gamma & = & 1 \\
(\xi,s) \Rightarrow_M \gamma & = & (\xi \Rightarrow_M \gamma) \times \textbf{\textit{Expr}}_M(s,\gamma)
\end{array}$$

Note that substitutions, σ may have their source scopes restricted, $\theta\sigma$ and their target scopes thinned $\sigma\theta$. Note that if $\theta \in [s] \leq \xi$ and $\sigma \in \xi \Rightarrow_M \gamma$, then the singleton restriction $\theta\sigma$ amounts to a projection into $\textbf{\textit{Expr}}_M(s,\gamma)$.

I shall routinely drop them M subscript where it is used uniformly.

Definition 3.5 (Weakening for Substitution, Identity Substitution). *If $\sigma : \xi \Rightarrow \gamma$, then let $\sigma\!\uparrow\! s$, its* weakening, *be given thus:*

$$\sigma\!\uparrow\! s = (\sigma(\vec{1},0), \#(\vec{0},1)) : (\xi,s) \Rightarrow (\gamma,s)$$

Iterating over scopes, we obtain

$$\iota_\gamma = ()|\gamma : \gamma \Rightarrow \gamma$$

mapping each variable to itself.

Definition 3.6 (Action of Substitution). *Substitutions act on expressions $t\sigma$ and postcompose with other substitutions $\rho\sigma$. This action is the identity on atoms and structural on pairs. We further have $(\#\theta)\sigma = \theta\sigma$ and we go under a binding at sort s by weakening $(x.t)\sigma = x.t(\sigma\!\uparrow\! s)$. Moreover, routine calculations show that substitutions are thus morphisms of a* category *acting* functorially *on expressions.*