Input. Subject. Output.

ANONYMOUS AUTHOR(S)

ACM Reference format:
Anonymous Author(s). 2022. Input. Subject. Output.. 1, 1, Article 1 (July 2022), 5 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.

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

I have specified this syntax in an informal 'scoped Backus-Naur form' notation. Each sort yields a nonterminal symbol carrying a scope, e.g. $\textsc{chk}(\gamma)$. 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/2022/7-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

$\textsc{jud}(\gamma)$	$::=$	$\textsc{chk}(\gamma) \ni \textsc{chk}(\gamma)$	checking a prior type
	$\mid$	$\textsc{syn}(\gamma) \in \textsc{chk}(\gamma)$	synthesizing a posterior type
	$\mid$	$\textsc{chk}(\gamma) = \textsc{chk}(\gamma)$	type equality
	$\mid$	$x \in \textsc{chk}(\gamma) \vdash \textsc{jud}(\gamma, \textsc{syn}\ x)$	local extension

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

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

Meanwhile, for synthesis, we have

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

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

$$\frac{}{T = T}$$

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.

- $\dfrac{e \in S \quad S = T}{T \ni \underline{e}}$ 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!
- $\dfrac{\star \ni S \quad x \in S \vdash \star \ni T}{\star \ni \prod S\ x.T}$ 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.
- $\dfrac{x \in S \vdash T \ni t}{\prod S\ x.T \ni \lambda\ x.t}$ We know $\star \ni \prod S\ x.T$, 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.
- $\dfrac{f \in \prod S\ x.T \quad S \ni s}{f\ s \in \{x \mapsto s\ :\ S\}T}$ The first premise promises us $\star \ni \prod S\ x.T$, 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 $\star \ni \{x \mapsto s\ :\ S\}T$.

We have established the positive counterpart to garbage-in-garbage-out! 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.

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

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.

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

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

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.

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

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

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

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.