Under consideration for publication in J. Functional Programming

The types who say \ni

CONOR T. MCBRIDE
University of Strathclyde
(*e-mail:* conor.mcbride@strath.ac.uk)

1 Introduction

This paper is about my religion. It introduces a discipline for constructing and validating bidirectional type systems, illustrated with a nontrivial running example — a bidirectional reconstruction of Per Martin-Löf's small and beautiful, but notoriously inconsistent dependent type theory from 1971 (Martin-Löf, 1971). 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 & Turner, 2000).

From the outset, it would seem prudent to manage expectations. I take the view that types are not inherent in things, but rather imposed on them by human design in arbitrary ways. Meaning is made, not found. A practical consequence of this viewpoint is that we may fix a generic syntax, simple and flexible, before giving any thought to the design of types and the meaningful forms of computation they justify. Every self-respecting religion pinches the parts it likes from other religions, so I will choose LISP s-expressions (McCarthy, 1960) as the generic syntax, but tighten up the treatment of variable binding with methods of de Bruijn (de Bruijn, 1972), distinguishing variables from atoms and naming them only in the interests of informal civility. One should not expect things expressible in this syntax to make sense: rather we should design ways to make sense of some of them. We get out what we put in, so let us seek understanding of how to direct our freedom towards virtuous outcomes.

Oft sought properties of type systems, such as 'stability under substitution' and 'subject reduction', are not to be had for the proving, but rather by construction in accordance with good guidance. The prospectus of this paper is to develop a metametatheory in which to ensure the good metatheoretic properties of a whole class of theories.

2 Unpacking the problem

By way of motivating an alternative approach, let us briefly examine the current situation. What are the problematic consequences of type synthesis? What are the obstacles to adopting a mixture of type checking and type synthesis? What makes subject reduction hard to prove? The ways in which we address these questions give us keys to their answers.

2.1 Which types must we write?

In order to ensure that types can be synthesized, we will need to write type annotations in some places. We work in a setting where types involve computation, so it is clear we would have to solve an ambiguous, let alone undecidable class of unification problems to omit all the type information in the Milner manner. We cannot sustain a type system on the manifestly false promise that functions are in general invertible. Our programs need strategically placed type information to supply components we cannot hope to construct in any other reliably effective way. The examples of dependent functions and pairs allow us to contrast fantasy with reality.

	fantasy	**reality**
functions	$\dfrac{x{:}S \vdash t \ : \ T}{\lambda x.t \ : \ (x{:}S) \to T}$	$\dfrac{\text{TYPE}\, S \quad x{:}S \vdash t \ : \ T}{\lambda x{:}S.t \ : \ (x{:}S) \to T}$
pairs	$\dfrac{s \ : \ S \quad t \ : \ T[s/x]}{(s,t) \ : \ (x{:}S) \times T}$	$\dfrac{s \ : \ S \quad x{:}S \vdash \text{TYPE}\, T \quad t \ : \ T[s/x]}{(s,t)_{x.T} \ : \ (x{:}S) \times T}$

In the case of functions, the domain of an abstraction must come from somewhere, and it must be in place and checked to be a type before it is reasonable to extend the context and synthesize the type of the body. However, once the body's type has been found, with respect to a generic variable, there is no choice about how to abstract that variable to yield a function type. In practice, one can place a metavariable in the context as x's type and hope to collect constraints on it from the way x is used, but one cannot expect that those constraints will yield a clear solution.

The situation is, if anything, worse in the case of pairs. While the type, S, of the first component, s, is clearly obtainable by synthesis, the type, $T[s/x]$, of the second component, t, yields but one instance of the pattern of dependency expressed by the pair type, from which we must abstract some $T[x]$. Substitution is not uniquely invertible. More concretely, the pair $(3,[0,1,2])$ of a number and a length-indexed list can be given any pair type where the length is computed by a function on the natural numbers which maps 3 to 3: there are a great many such functions. There is no choice but to give this function explicitly.

We can construct the real rules from the fantasy rules, determining which annotations are mandated by magical misapprehensions. That is to say, it is not good enough to write schematic variables in typing rules without ensuring a clear source for their instantiation. We might profit from a finer analysis of scope and information flow in typing rules, giving each schematic variable one binding site and zero or more use sites. This paper will deliver one such analysis in due course. However, we can already see that the necessary annotations will arise from the specifics of the types in question, rather than in a uniform way that lends itself to a generic methodology of metatheory.

But it gets worse. If we transform a dependent *telescope*,

$$x_0 : S_0, x_1 : S_1[x_0], \ldots, x_n : S_n[x_0, \ldots, x_{n-1}]$$

into a right-nested pair type, the corresponding values will be festooned with redundant but differently instantiated copies of the telescope's tails.

$$\begin{array}{l}(s_0, (s_1, \ldots (s_{n-1}, s_n)_{x_{n-1}.S_n[s_0,\ldots,s_{n-1}]} \\ \quad\quad \cdots)_{x_1.(x_2\colon S_2[s_0,x_1])\times\ldots S_n[s_0,x_1,\ldots,x_{n-1}]} \\ \quad)_{x_0.(x_1\colon S_1[x_0])\times\ldots S_n[x_0,x_1,\ldots,x_{n-1}]}\end{array}$$

The insistence that every subterm carry all the information necessary for the synthesis of its type, regardless of where it sits and how much we might already know of our *requirements* for it, leads to a corresponding blow up. The core languages of both the Glasgow Haskell Compiler and the Coq proof assistant are presently afflicted: Garillot's thesis documents a situation where an apparently sensible approach to packaging mathematical structures is prevented in practice by an exponential growth in redundant type information. This must stop!

2.2 Can we turn things around?

The irony of the type annotation problem, above, is that the 'fantasy' rules make perfect sense when seen as type *checking* rules. A function type tells you the domain type which goes in the context and the range type which checks the body of an abstraction. A pair type tells you the general scheme of dependency which must be instantiated when checking components. Might we not propagate our requirements through the structure of terms, rather than requiring each individual subterm to be self-explanatory?

Such an approach was pioneered by Pierce and Turner under the name 'Local Type Inference', and has since been explored by many others, notably by Dunfield in the otherwise troublesome setting of intersection types. It is indeed much easier to see that you get what you want if you know what you want in advance. The usual situation is that one checks types for introduction forms and synthesizes types for elimination forms. Everything synthesizable is also checkable, as this situation gives *two* candidates for the type of the term in question whose consistency can then be tested. However, there is no hope to synthesize types for terms which are merely checkable, as the number of candidate types is *zero*. The latter bites when we seek to express a β-redex — the elimination of an introduction form:

$$(\lambda x.t)\, s$$

Even if we are given the type of this expression, we cannot hope to compute the type at which to check the abstraction, inferring the general pattern of dependency from one instance of it.

The standard remedy is to restrict the language to β-normal forms and conceal all opportunities behind *definitions*. We may give a definition

$$\mathsf{f} : (x\colon S) \to T;\ \mathsf{f} = \lambda x.t$$

where the type annotation is no mere act of pious documentation but rather our assurance that the types of all identifiers in scope are known. It is then straightforward to synthesize a type for the application $\mathsf{f}\, s$ — or is it? That type would be $T[s/x]$ for some suitable notion of substitution, but the textual replacement of x by s will not preserve β-normality: substitution can create redexes.

The spider we usually swallow to catch this fly is *hereditary* substitution, which contracts all the redexes introduced by the replacement of variables, and all the further redexes induced by those contractions, and so on, until we have restored β-normality. The efficacy of this solution rests on hereditary substitution being well defined, which amounts to showing that our calculus is β-normalizing. For systems with weak function spaces, such as the logical frameworks from whose literature the technique originates, this is no harder than normalizing the simply typed λ-calculus. In the general setting of dependent type theories, however, we have not such an easy victory: for Martin-Löf's inconsistent 1971 type theory, hereditary substitution is *not* well defined, but the system enjoys subject reduction, none the less.

In the business of establishing metatheoretic properties of type systems, it is certainly preferable if basic hygiene properties sych as subject reduction can be established more cheaply than by appeal to as heavy a requirement as normalization. Indeed, we have only a chance of showing that well typed terms are normalizing, so it approaches the circular to rely on normalization in the definition of what it means to be well typed in the first place.

Even in settings where hereditary substitution is not well defined, one might consider presenting it relationally, refining the burden of proof to individual cases. The application rule would become

$$\frac{f \ : \ (x\!:\!S) \to T \quad s \ : \ S \quad T[s] \Downarrow T'}{f\, s \ : \ T'}$$

yielding a derivation only where the substitution is successful. The trouble here is that the relation $T[s] \Downarrow T'$ is manifestly not stable under substitution — instantiating free variables can cause inert terms to compute, perhaps for ever.

The effective remedy is to ensure that computation has a small-step presentation which is stable under substitution. We must ensure that our syntax is capable of expressing β-redexes, and to that end, let us introduce type annotations which mediate between introduction and elimination forms, ensuring that we have enough information to validate them. In what follows, we shall do so *uniformly*, rather than placing annotations differently for different types. The purpose of a type annotation is exactly to characterize a redex, so it will help to standardize the ways in which redexes arise.

2.3 What makes subject reduction difficult to prove?

We might very well hope to prove the following admissible

$$\frac{\Gamma \vdash t \ : \ T \quad t \leadsto t'}{\Gamma \vdash t' \ : \ T} \qquad \frac{\Gamma \vdash \textsc{type}\, T \quad T \leadsto T'}{\Gamma \vdash \textsc{type}\, T'}$$

and we should be mortified were it not the case. However, this statement does not follow by induction on the typing derivation for the subject, t. In dependent typing derivation, components from the term being checked can be copied to the right of the colon (e.g., in the application rule) and, when moving under binders (e.g., when checking that a function type is well formed), into the context. The above statement allows for no computation in the context or the type, so our induction hypotheses may fail to cover the cases which arise. Consider the case for

$$\textsc{type}\ (x\!:\!S) \to T \qquad (x\!:\!S) \to T \leadsto (x\!:\!S') \to T$$

where the derivation is by the rule

$$\frac{\textsc{type}\ S \quad x\!:\!S \vdash \textsc{type}\ T}{\textsc{type}\ (x\!:\!S) \to T}$$

We will have an induction hypothesis concerning reduction of T after the derivation of

$$\Gamma, x\!:\!S \vdash \textsc{type}\ T$$

but the computation in the type means that we now need to show

$$\Gamma, x\!:\!S' \vdash \textsc{type}\ T$$

with a new context about which we know too little.

We shall certainly need a more general formulation of subject reduction in which things other than the subject — contexts and types — may also compute, but this exposes us to a further risk in cases where the typing rules move information from context and type back into the subject position. E.g., the conversion rule is sometimes formulated as

$$\frac{t\ :\ S \quad S \cong T \quad \textsc{type}\ T}{t\ :\ T}$$

where T is β-convertibility: as reverse β-steps are most certainly not guaranteed to preserve types, we must confirm that T makes sense. If our formulation of subject reduction allows too much computation in the type T, our induction hypothesis for $\textsc{type}\ T$ may be too weak to show that we still have a meaningful type.

In summary, the formulation of subject reduction statements is extremely sensitive to how much computation is permitted in which places, and the literature of metatheory for dependent type systems shows considerable delicate craft. What design principles might we follow to be sure of a robust proof strategy? Read on!

3 What is to be done?

The prospectus I offer is a *general* proof of subject reduction for a large class of dependent type theories, resting only on conditions which can be checked mechanically. That is, for the theories in this class, subject reduction is had for the asking.

In order to obtain this result I shall need to develop disciplines for specifying type theories which, by design, avoids pitfalls like those outlined above. In some cases, these disciplines will merely make explicit what is, in any case, standard practice. In other cases, I deviate from the approach usually found in the presentation of type synthesis systems to exploit particular characteristics of the bidirectional setup.

Central to the project is a careful analysis of the roles each position plays in the judgement forms and the flow of information through typing rules. The key idea is that a rule is a *server* for derivations of its conclusion and a *client* for derivations of its premise. A judgement form is thus a tiny little session type, specifying the protocol for these client-server interactions. We thus may thus formulate a clearer policy of who is promising what to whom and check whether rules are compliant — we do not write down any old nonsense.

This tighter grip on information flow will manifest itself in a separation of the kinds of formula we may write in a rule into *patterns*, which contain the binding sites of the schematic variables, and *expressions*, which may contain substitutions on schematic variables. An immediate consequence is that rules can never demand the magical inversion of substitions. A more subtle consequence is that the typing assumptions we encounter can always be inverted mechanically to determine what is known about each schematic variable. A careful policing of the scope of schematic variables, particularly those which occur in the subjects of judgements, will enable us to formulate the statement of subject reduction in a way that guarantees the effectiveness of induction on typing derivations. Likewise, a tight Trappist discipline on free variables in rules will ensure that any expressible system of rules is stable under substitution.

I propose to work in a generic syntax, adequate to express canonical constructions which allow the binding of variables, with a uniform treatment of elimination and thus exactly one way to construct a redex, exposing the type at which reduction can happen. The patterns and expressions which may appear in rules will be specified with respect to this syntax. Redexes, too, will be characterised in terms of patterns: for any canonical type, we shall be able to compute its set of non-overlapping redexes which must be given reducts to ensure progress, yielding a rewriting system which is confluent by construction. That each reduct have the same type as its redex will be the key condition for subject reduction — unsurprisingly necessary, but remarkably sufficient for any protocol-compliant system of rules.

4 Sufficient syntax

Formally, I work with a generic de Bruijn-style nameless syntax, with syntactic categories indexed by their scope. The embedding of one scope into another is called a *thinning*. This section defines the syntax and the actions upon it of thinnings and substitutions.

4.1 Scoped and separated syntactic classes

For the benefit of human beings, a *scope* is written as a list of identifiers $x_{n-1}, \ldots x_0$, although the inhuman truth is that it is just the number n which gives the length of the sequence. I refer to scopes by metavariables γ and δ, with ε denoting the empty scope. A *variable* is an identifier x_i selected from a scope, serving as the human face of its index i. I specify grammars relative to an explicit scope, γ and write x_γ to mean 'a variable from γ'.

Unlike variables, *atoms* (a, A) really are named global constants which play the role of tags — their purpose is not to stand for things but to be told apart. The symbol () is considered an atom and pronounced 'nil'.

Definition 1 (*constructions and computations, essential and liberal*)

The object-level syntax is specified, written as a subscript, and is divided into four distinct mutually defined grammatical classes, each with standard metavariable conventions, arranged as four quadrants thus:

$d \backslash l$	**essential**	**liberal**
construction	k_γ, K_γ $::=\ a$ $\mid\ (s_\gamma . t_\gamma)$ $\mid\ \backslash x\, t_{\gamma,x}$	$s_\gamma, t_\gamma, S_\gamma, T_\gamma$ $::=\ k_\gamma$ $\mid\ [n_\gamma]$
computation	n_γ, N_γ $::=\ x_\gamma$ $\mid\ e_\gamma\, s_\gamma$	$e_\gamma, f_\gamma, E_\gamma, F_\gamma$ $::=\ n_\gamma$ $\mid\ (t_\gamma : T_\gamma)$

Let us write $\mathbf{Term}(l,d,\gamma)$ for the set of terms in the quadrant given by l and d with scope γ.

The meaningfulness of *constructions* will always be *checked*, relative to some prior expectation. The *essential* constructions give us the raw materials for canonical values, and they will always be invariant under computation. Let us adopt the LISP convention that matching parentheses preceded by a dot may be deleted, along with the dot, which is conveniently prejudicial to right-nested, nil-terminated lists. Our constructions will often be such lists, with an atom at the head. For example, dependent function types in γ will be constructions of form

$$(\Pi S \backslash x T) \quad \text{which abbreviates} \quad (\Pi.(S.(\backslash x T.())))$$

where Π is a particular atom, S is a γ-construction and T is a γ,x-construction. It is not my habit to notate explicitly the potential dependency of T on x: the abstraction makes that much clear.

The *liberal* constructions extend the canonical constructions with *thunks* — essential computations which have not yet achieved canonical form, either because they have not yet reduced, or because a variable is impeding reduction.

Meanwhile, the *computations* will always admit a type synthesis process. The *essential* constructions comprise variables (whose type will be assigned by a context) and eliminations, overloading the application syntax — the computation e to the left is being eliminated, and its synthesizable type will tell us how to check that the construction s to its right is a valid eliminator. These two give us the traditional forms of *neutral* term.

The *liberal* computations extend the neutral computations with *radicals*, in the chemical sense, being canonical forms with a type annotation which gives the information needed to check both the canonical form it annotates and the eliminator it is 'reacting' with. If we exclude radicals from this syntax, we obtain the *normal forms*. Radicals are permitted only in eliminations, and these are the only eliminations which compute. That is, there is no computation step which proceeds uninformed by the type at which computation is happening.

Thunks and radicals form the connection between constructions and computations, but you will notice that the syntax carefully precludes the thunking of a radical — a computation which is eliminated no further needs no type annotation. We may extend thunking to liberal computations as a ‘smart constructor’ which deletes the annotations of radicals.

$$[(t : T)] = t$$

Observe also that we may now lift *all* the term formers to act on liberal components, yielding liberal results, for everything apart from thunks admits liberal substructures. In particular, it becomes reasonable to substitute a liberal computation for a variable, in either a construction or a computation, yielding a liberal construction or computation, respectively. I write t/e for such a substitution in a construction, when it is clear from context which variable in t is being substituted.

With this apparatus in place, we may, for example, reformulate the traditional β-rule for functions as a rewriting rule on liberal computations, thus:

$$(\backslash x\, t : (\Pi\, S\ \backslash x\, T))\ s \rightsquigarrow (t/(s : S) : T/(s : S))$$

This rule substitutes a radical for the bound variable, creating potential redexes wherever that variable is eliminated. Moreover, the whole reduct will be radical. The type annotations thus mark the active computation sites and are discarded whenever computation concludes.

The β-normal forms are characterised by replacing $(t_\gamma : T_\gamma)$ in the grammar by $(n_\gamma : T_\gamma)$, ensuring that all radicals are inert because of some free variable. We must resist the clear temptation to elide type annotations on neutral terms during reduction as the property of being neutral is not stable under substitution.

4.2 The category of thinnings acts functorially on syntax

Definition 2 (*thinning*)
A thinning is an order preserving embedding between scopes

$$\theta : \gamma \sqsubseteq \delta$$

I typically use θ, ϕ and ψ as metavariables for thinnings. The thinnings are generated by

$$\frac{}{\varepsilon : \varepsilon \sqsubseteq \varepsilon} \qquad \frac{\theta : \gamma \sqsubseteq \delta}{\theta 0 : \gamma \sqsubseteq \delta, x} \qquad \frac{\theta : \gamma \sqsubseteq \delta}{\theta 1 : \gamma, x \sqsubseteq \delta, x}$$

That is, thinnings arise in the manner of Pascal’s triangle: there are $\binom{m}{n}$ thinnings from n to m, which is not surprising, as they correspond to selections.

A thinning is given as a bit vector the length of its target scope with its source scope being the ‘population count’, i.e., number of 1s, in the vector. Thinnings form a well known category: the *semi-simplicial* category, often notated Δ_+.

Definition 3 (*identity thinning*)
The identity thinning $\mathbf{1}_\gamma : \gamma \sqsubseteq \gamma$ is given by

$$\mathbf{1}_\varepsilon = \varepsilon \qquad \mathbf{1}_{\gamma, x} = \mathbf{1}_\gamma 1$$

Informally, we may just write $\mathbf{1}$. The empty thinning, $\mathbf{0}$ is generated analogously, repeating 0 to the appropriate length.

Definition 4 (*composition of thinnings*)

If $\theta : \gamma \sqsubseteq \gamma'$ and $\phi : \gamma' \sqsubseteq \gamma''$, then $\theta;\phi : \gamma \sqsubseteq \gamma''$ defined as follows:

$$\varepsilon;\varepsilon = \varepsilon \qquad \theta;\phi 0 = (\theta;\phi)0 \qquad \theta b;\phi 1 = (\theta;\phi)b$$

Definition 5 (*weakening*)
The thinning $\iota 0 : \gamma \sqsubseteq \gamma, x$ which adds one new local variable is written $\uparrow$, and we denote its action on a thing by $\hat{\cdot}$. In particular, the weakening of a thinning, $\hat{\theta}$ is given by $\theta;\uparrow$.

Lemma 6 (*category of thinnings*)
We have the usual categorical laws:

$$\mathbf{1};\theta = \theta = \theta;\mathbf{1} \qquad (\theta;\phi);\psi = \theta;(\phi;\psi)$$

Proof
Functional induction on the (graph of) composition readily establishes these results. □

Lemma 7 (*functoriality of scope extension*)
The actions $\cdot, x$ on scopes and $\cdot 1$ on thinnings is functorial:

$$\mathbf{1}_{\gamma} 1 = \mathbf{1}_{\gamma,x} \qquad (\theta;\phi)1 = (\theta 1;\phi 1)$$

Proof
This holds by definition. □

Remark 8 (*thinnings as an integer monoid*)
In another life, I teach undergraduates about computer hardware. Consequently, I recognize the identity thinning as the two's complement representation of -1. Effectively, we may regard the integers as the infinite right-to-left bit vectors which eventually stabilise as all 0 (for non-negative integers) or all 1 (for negative integers). Thinning composition induces a monoid on the integers whose neutral element is -1. The details are left to the curious reader.

Meanwhile, thinnings form a monoidal category by concatenation γ, γ' of scopes and θ, θ' of thinnings.

The formal truth is that a 'variable' x_γ is a thinning in some $\varepsilon, x \sqsubseteq \gamma$, i.e., from the singleton scope to γ. Noting that γ occurs positively in the grammar, we obtain that each of our four syntactic quadrants is a functor from thinnings to **Set**.

Definition 9 (*action of a thinning*)
If $\theta : \gamma \sqsubseteq \delta$, it has a quadrant-preserving action, $\cdot\theta : \mathbf{Term}(l,d,\gamma) \to \mathbf{Term}(l,d,\delta)$

$$\begin{array}{rcl@{\qquad}rcl}
a\theta & = & a & [n]\theta & = & [n\theta] \\
(s.t)\theta & = & (s\theta.t\theta) & & & \\
\left(\backslash x\, t\right)\theta & = & \backslash x\left(t\left(\theta 1\right)\right) & & & \\
x\theta & = & x;\theta & (t:T)\theta & = & (t\theta : T\theta) \\
\left(e\, s\right)\theta & = & \left(e\theta\right)\left(s\theta\right) & & &
\end{array}$$

Lemma 10 (*functoriality of the thinning action*)
The thinning action extends $\mathbf{Term}(l,d,\cdot)$ to a functor from Δ_+ to **Set**.

Proof

We must show that $m\mathbf{1} = m$ and that $m(\theta;\phi) = m\theta\phi$ for any term m in any quadrant. This is established straightforwardly by induction on m, relying on the functoriality of scope extension to pass under an abstraction. □

4.3 The category of substitutions acts functorially on syntax

Let us turn now to the matter of *substitutions*, which act simultaneously on all variables in a scope. Let us take σ and ρ as metavariables for substitutions.

Definition 11 (*substitution*)
Let $\gamma \Rightarrow \delta$ be the set of substitutions from γ variables to δ terms, i.e., mappings

$$\sigma : (\varepsilon, x \sqsubseteq \gamma) \to \mathbf{Term}(\mathbf{liberal}, \mathbf{computation}, \delta)$$

Let us write $x\sigma$ for the image of some x. We may write ε for the trivial substitution from the empty scope. If $\sigma : \gamma \Rightarrow \delta$ and $e : \mathbf{Term}(\mathbf{liberal}, \mathbf{computation}, \delta)$, then we say

$$\sigma, e : \gamma, x \Rightarrow \delta \qquad x(\sigma, e) = e \qquad y(\sigma, e) = y\sigma \text{ if } x \neq y$$

Formally, a substitution is a vector of terms, acting on de Bruijn indices by projection.

Note that as variables are computations, so must be their images, but that we shall permit those images to be liberal, allowing in particular the replacement of variables by radicals. We shall establish that substitutions form a category in due course. Let us first see how they operate. Before we can give their action on terms, we say how they pass under binders.

Definition 12 (*weakening of substitutions*)
If $\sigma : \gamma \Rightarrow \delta$, then we may define its weakening, $\sigma 1 : \gamma, x \Rightarrow \delta, x$, by

$$\sigma 1 = \hat{\sigma}, x \quad \text{where} \quad \hat{\sigma} = \sigma;\uparrow \quad \text{where} \quad y\left(\sigma;\theta\right) = \left(y\sigma\right)\theta$$

That is, $\sigma;\theta$ denotes pointwise action of θ on terms in σ.

Definition 13 (*action of a substitution*)
If $\sigma : \gamma \Rightarrow \delta$, it has a liberalising action, $\cdot\sigma : \mathbf{Term}(l, d, \gamma) \to \mathbf{Term}(\mathbf{liberal}, d, \delta)$, extending its action on variables to terms.

$$\begin{array}{rcl@{\qquad}rcl}
a\sigma & = & a & [n]\sigma & = & [n\sigma] \\
(s.t)\sigma & = & (s\sigma.t\sigma) & & & \\
\left(\backslash x t\right)\sigma & = & \backslash x\left(t\left(\sigma 1\right)\right) & & & \\
\left(e\, s\right)\sigma & = & \left(e\sigma\right)\left(s\sigma\right) & (t : T)\sigma & = & (t\sigma : T\sigma)
\end{array}$$

Note that if $x\sigma = (t : T)$, then $[x]\sigma = [(t : T)] = t$, as the smart $[\cdot]$ strips the type annotation.

Definition 14 (*postcomposition by substitution*)
If $\theta : \gamma_0 \sqsubseteq \gamma_1$ and $\sigma : \gamma_1 \Rightarrow \gamma_2$, their composition *selects* θ's domain from σ.

$$\theta;\sigma : \gamma_o \Rightarrow \gamma_2 \qquad x\left(\theta;\sigma\right) = \left(x\theta\right)\sigma$$

If $\rho : \gamma_0 \Rightarrow \gamma_1$ and $\sigma : \gamma_1 \Rightarrow \gamma_2$, their composition makes σ act *pointwise* on terms in ρ.

$$\rho;\sigma : \gamma_o \Rightarrow \gamma_2 \qquad x\left(\rho;\sigma\right) = \left(x\rho\right)\sigma$$

Lemma 15 (*action of compositions*)
All four compositions of thinnings and substitions act on terms as the sequence of the component actions.

$$t\left(\theta;\phi\right) = \left(t\theta\right)\phi \quad t\left(\theta;\sigma\right) = \left(t\theta\right)\sigma \quad t\left(\rho;\phi\right) = \left(t\rho\right)\phi \quad t\left(\rho;\sigma\right) = \left(t\rho\right)\sigma$$

Proof
Structural induction on terms establishes this property. To pass under binders, we must establish that weakening distributes $(\cdot;\cdot)1 = (\cdot 1;\cdot 1)$, which follows from the fact that

$$\hat{t}\left(\cdot 1\right) = t\hat{\cdot}$$

for both thinnings and substitutions. □

To complete the components for the category of substitutions, we must have the identity.

Definition 16 (*identity substitution*)
The identity substitution $\iota_\gamma : \gamma \Rightarrow \gamma$ is given by

$$\iota_\varepsilon = \varepsilon \qquad \iota_{\gamma,x} = \iota_\gamma 1$$

A straightforward induction shows that $x\iota = x$.

Lemma 17 (*category of substitutions*)
Substitutions $\sigma : \gamma \Rightarrow \delta$ are the arrows of a category with identity ι and composition ;. Moreover, scope extension, $\cdot,x$ is an endofunctor acting on arrows as $\cdot 1$, and action on terms makes $\mathbf{Term}(\mathbf{liberal}, d, \cdot)$ a functor from substitutions to **Set**.

Proof
The category and functor laws collate the above results about identity and composition of thinnings and substitutions. □

5 Judging judgements

A type theory is presented as a system of rules, each of which has zero or more premises and one conclusion, all of which are *judgements*.

The judgement forms are specified as sequences of *punctuation* and *places*. A judgement is written by putting *formulae* which stand for sets of terms into the *places*. Each place in a judgement form is assigned a *mode*.

Definition 18 (*mode*)
There are three modes which may be assigned to a place in a judgement:

input an input is a term supplied by a rule client — this term is already in some sense trusted;
subject a subject is a term supplied by a rule client — ‘trust’ in this term remains to be established by appeal to the rule;
output an output is a term supplied by a rule server — this term is guaranteed to be ‘trustworthy’.

What does it mean to be 'trusted'? For each input place in a judgement form, we must specify a judgement where that input now stands as a subject — such a judgement is called a *precondition* and represents a proof obligation to be discharged by a rule client. For each output place in a judgement form, we must specify a judgement where that output now stands as a subject — such a judgement is called a *postcondition* and represents a proof obligation to be discharged by a rule server.

Without further ado, let me specify the judgement forms I propose. Judgements exist in some scope, γ, and I shall be careful to give scopes to their places. I shall draw a box around the subjects, of which there will be at most one, and ensure that inputs stand left of the box, with outputs to the right. I write preconditions in braces to the left of the judgement form, with post conditions in braces to the right.

Definition 19 (*γ-judgement*)
The judgements, J_γ, in scope γ are given inductively as follows:

$\mathbf{Pre}(J_\gamma)$	J_γ	$\mathbf{Post}(J_\gamma)$	purpose
$\{\}$	$\textsc{type}\ \boxed{T_\gamma}$	$\{\}$	type construction
$\{\textsc{type}\ T_\gamma\}$	$T_\gamma \ni \boxed{t_\gamma}$	$\{\}$	type checking
$\{\}$	$\boxed{e_\gamma} \in S_\gamma$	$\{\textsc{type}\ S_\gamma\}$	type synthesis
$\{\}$	$x_\gamma\ :\ S_\gamma$	$\{\textsc{type}\ S_\gamma\}$	type lookup
$\{\textsc{type}\ S_\gamma, \textsc{type}\ T_\gamma\}$	$S_\gamma = T_\gamma \square$	$\{\}$	type equality
$\{\textsc{type}\ S_\gamma, x\!:\!S_\gamma \vdash \mathbf{Pre}(J'_{\gamma,x})\}$	$x\!:\!S_\gamma \vdash J'_{\gamma,x}$	$\{x\!:\!S_\gamma \vdash \mathbf{Post}(J'_{\gamma,x})\}$	context extension

If you are familiar with presentations of type theories, you will immediately notice the omission of explicit *contexts*. This is not mere laziness on my part, but is done with a purpose that I shall shortly reveal. Typing γ-rules conclude one γ-judgement from zero or more γ-judgement premises, where all of these γ-judgements implicitly share the same γ-context. Typing rules may *locally* extend the context, assigning a valid type to the new most local variable.

Definition 20 (*γ-context, context validity, global judgements*)
Let us say when Γ is a syntactically well formed γ-context, and when it is, moreover, *valid*. If so, we may write $\Gamma \vdash J_\gamma$ to assert the *global judgement* that a particular γ-judgement holds in Γ.

- The only ε-context is ε, and it is valid.
- If Γ is a γ-context, then $\Gamma, x\!:\!S_\gamma$ is a γ, x context, valid whenever Γ is valid and $\Gamma \vdash \textsc{type}\ S_\gamma$.

It may seem peculiar that context validity is not presented as a judgement. This, also, is purposeful: it prevents the *re*validation of the context by typing rules. I thus depart from standard practice from taking the validity of the empty context as the only axiom, where the rules which concern atomic or variable subjects take context validity as a premise. In the client-server analysis of typing rules, we may consider context validity to be entirely the responsibility of the client: garbage in, garbage out! The precondition for the context extension judgement form does exactly the work required to ensure that the extended context is valid if the original context is.

There is but one rule to derive context extension judgements, and it must be given globally.

Definition 21 (*context extension*)

$$\textsc{extn}\ \frac{\Gamma, x\!:\!S \vdash J}{\Gamma \vdash x\!:\!S \vdash J}$$

The client invoking the above rule must ensure that Γ is valid and that $\Gamma \vdash \textsc{type}\ S$, which is sufficient to ensure the validity of the context in the premise.

Meanwhile, we have two global rules for looking up the type of a variable in a context.

Definition 22 (*type lookup*)

$$\textsc{top}\ \frac{}{\Gamma, x\!:\!S \vdash x\ :\ \hat{S}} \qquad \textsc{pop}\ \frac{\Gamma \vdash x\ :\ S}{\Gamma, y\!:\!T \vdash x\ :\ \hat{S}}\ x \neq y$$

The side condition in the latter is for human consumption only: formally, it is clear which context entry a de Bruijn index indicates. Note, however, that we incur a proof obligation in respect of these rules. In each case, we are promised (by the client in the former, by the server of the premise in the latter), that $\Gamma \vdash \textsc{type}\ S$, but the postcondition for type lookup judgements requires us to show that $\Gamma, y\!:\!T \vdash \textsc{type}\ \hat{S}$. To have any chance of that, we need type construction to be stable under thinning in general. I propose to achieve that basic and useful property once and for all, by means of two self-denying ordinances.

Firstly, no other rules may be presented in terms of global judgements: all must be *locally* presented in terms of γ-judgements for an arbitrary γ, and interpreted for global judgements by globalisation with respect to an arbitrary shared γ-context, Γ. Secondly, the only rule which may use type lookup in a premise is the following:

Definition 23 (*variable type synthesis*)

$$\textsc{var}\ \frac{x\ :\ S}{x \in S}$$

That is the purpose for omitting the global context in typing rules: we obtain the stability of judgements with respect to actions on free variables by ensuring that the rules talk about only those parts of the syntax which are unaffected by those actions.

Dogma 24 (*You do not talk about free variables.*)
No free variable may ever appear in any typing rule but VAR, TOP and POP. All other rules are expressed in terms of ε-judgements, which embed into γ-judgements by $\mathbf{0}$, as follows.

Definition 25 (*thinning judgements*)
If J is a γ-judgement and $\theta : \gamma \sqsubseteq \delta$, then $J\theta$ is the δ-judgement given by the action of θ distributed structurally to all the places of J. I give only the case for context extension

$$\left(x\!:\!S \vdash J\right)\ \theta = x\!:\!S\theta \vdash J\ \left(\theta 1\right)$$

as the rest just propagate θ unchanged.

The only variables which appear in the rest of the rules are *bound*, either by the abstraction construct of the syntax, or by the context extension judgement form, and thus It will take a little more clarity about the structure of rules to achieve this outcome for substitution, but let us deal with the case for thinnings now.

Definition 26 (*contextual thinning*)
If $\theta : \gamma \sqsubseteq \delta$, Γ is a γ-context and Δ is a δ-context then $\Gamma \sqsubseteq_\theta \Delta$ asserts that θ extends from scopes to contexts and holds as follows.

$$\frac{}{\varepsilon \sqsubseteq_\varepsilon \varepsilon} \qquad \frac{\Gamma \sqsubseteq_\theta \Delta}{\Gamma, x\!:\!S \sqsubseteq_{\theta 1} \Delta, x\!:\!S\theta} \qquad \frac{\Gamma \sqsubseteq_\theta \Delta \quad \Delta \vdash \textsc{type}\ T}{\Gamma \sqsubseteq_{\theta 0} \Delta, y\!:\!T}$$

Lemma 27 (*stability under thinning*)
The following is admissible

$$\frac{\Gamma \sqsubseteq_\theta \Delta \quad \Gamma \vdash J}{\Delta \vdash J\theta}$$

Moreover, if Γ is valid and $\Gamma \sqsubseteq_\theta \Delta$, then Δ is valid.

Proof
For type lookup judgements, we proceed by induction on the derivation of $\Gamma \sqsubseteq_\theta \Delta$ and inversion of $\Gamma \vdash x : S$: we must show $\Delta \vdash x;\theta : S\theta$. There are three cases:

- $\Gamma, x\!:\!S \sqsubseteq_{\theta 1} \Delta, x\!:\!S\theta$ and $\Gamma, x\!:\!S \vdash x : \hat{S}$ by TOP. Invoke TOP.
- $\Gamma, y\!:\!T \sqsubseteq_{\theta 1} \Delta, y\!:\!T\theta$ and $\Gamma, x\!:\!S \vdash x : \hat{S}$ by POP. Invoke POP.
- $\Gamma \sqsubseteq_{\theta 0} \Delta, y\!:\!T$. Invoke POP.

In each case, we rely on the fact that

$$\hat{S}\left(\theta 1\right) = S\left(\uparrow;\theta 1\right) = S\left(\theta;\uparrow\right) = \widehat{S\theta}$$

For the rest of the judgement forms, we proceed by induction on derivations. For $\Gamma \sqsubseteq_\theta \Delta$ and $\Gamma \vdash x\!:\!S \vdash J$, we need merely invoke the induction hypothesis for $\Gamma, x\!:\!S \sqsubseteq_{\theta 1} \Delta, x\!:\!S\theta$ and $\Gamma, x\!:\!S \vdash J$. For the VAR rule, we have already established the conformity of type lookup. For all the remaining rules, as yet unspecified, Dogma 24 insists that we have deduced some $\Gamma \vdash J\mathbf{0}$ from some $\Gamma \vdash J_i\mathbf{0}$. Inductively, we obtain $\Delta \vdash J_i\mathbf{0}\theta$, but $\mathbf{0};\theta = \mathbf{0}$, so we may invoke the very same rule to deduce $\Delta \vdash J_i\mathbf{0}$, and again, $\mathbf{0} = \mathbf{0};\theta$ so we have the induction conclusion. □

References

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

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

McCarthy, John. (1960). Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the acm*, **3**(4), 184–195.

Pierce, Benjamin C., & Turner, David N. (2000). Local type inference. *ACM trans. program. lang. syst.*, **22**(1), 1–44.