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 should 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	$\dfrac{x\!:\!S \vdash t \;:\; T}{\lambda x.t \;:\; (x\!:\!S) \to T}$	$\dfrac{\textsc{type}\,S \quad x\!:\!S \vdash t \;:\; T}{\lambda x\!:\!S.t \;:\; (x\!:\!S) \to T}$
pairs	$\dfrac{s \;:\; S \quad t \;:\; T[s/x]}{(s,t) \;:\; (x\!:\!S) \times T}$	$\dfrac{s \;:\; S \quad x\!:\!S \vdash \textsc{type}\,T \quad t \;:\; T[s/x]}{(s,t)_{x.T} \;:\; (x\!:\!S) \times T}$

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.

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

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 for computation behind *definitions*. We may give a definition

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

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 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 characterized 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:

$d \setminus l$	**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 *target* 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:

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

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 characterized 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,δ of scopes and θ,ϕ of thinnings. If $\theta : \gamma \sqsubseteq \gamma'$, we may write $\theta\delta$ for the concatentation $\theta,\mathbf{1}_\delta : \gamma,\delta \sqsubseteq \gamma',\delta$.

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, $\cdot\theta : \mathbf{Term}(l,d,\gamma) \to \mathbf{Term}(l,d,\delta)$

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

Note that when disambiguating terms, I am careful to use a form of bracketing, $\Big(\cdot\Big)$, which is not part of the syntax of terms, but happy to use ordinary parentheses to disambiguate other things, such as thinnings.

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. □

Before we leave the category of thinnings, let me establish one more property which will prove crucial to my treatment of patterns.

Lemma 11 (Δ_+ *has pullbacks*)
If $\theta_i : \gamma_i \sqsubseteq \gamma$ for $i = 0,1$, then there exists a scope δ and thinnings $\phi_i : \delta \sqsubseteq \gamma_i$ for $i = 0,1$ such that $phi_0;\theta_0 = \phi_1;\theta_1$ with the universal property that for any δ' and $\phi'_i : \delta' \sqsubseteq \gamma_i$ such that $phi'_0;\theta_0 = \phi'_1;\theta_1$, we have some $\psi : \delta' \sqsubseteq \delta$ such that $\phi'_i = \psi;\phi_i$ for $i = 0,1$. We define $\mathbf{pullback}(\theta_0,\theta_1) = (\phi_0,\phi_1)$, leaving δ implicit.

Proof
We shall see that δ is the part of γ selected by *both* θ_i, which needs must embed in both γ_i. More formally, we compute pullbacks thus:

$$\begin{array}{lcll}
\mathbf{pullback}(\varepsilon,\varepsilon) & = & (\varepsilon,\varepsilon) & \\
\mathbf{pullback}(\theta_0 0,\theta_1 0) & = & (\phi_0,\phi_1) & \text{where } (\phi_0,\phi_1) = \mathbf{pullback}(\theta_0,\theta_1) \\
\mathbf{pullback}(\theta_0 0,\theta_1 1) & = & (\phi_0,\phi_1 0) & \text{where } (\phi_0,\phi_1) = \mathbf{pullback}(\theta_0,\theta_1) \\
\mathbf{pullback}(\theta_0 1,\theta_1 0) & = & (\phi_0 0,\phi_1) & \text{where } (\phi_0,\phi_1) = \mathbf{pullback}(\theta_0,\theta_1) \\
\mathbf{pullback}(\theta_0 1,\theta_1 1) & = & (\phi_0 1,\phi_1 1) & \text{where } (\phi_0,\phi_1) = \mathbf{pullback}(\theta_0,\theta_1)
\end{array}$$

The proof of the universal property is a straightforward induction on the call graph of $\mathbf{pullback}(\cdot,\cdot)$. □

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 12 (*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 13 (*weakening of substitutions*)
If $\sigma : \gamma \Rightarrow \delta$, then we may define its weakening, $\sigma 1 : \gamma, x \Rightarrow \delta, x$, by

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

That is, $\sigma;\theta$ denotes pointwise action of θ on terms in σ. We may define $\sigma\gamma' : \gamma, \gamma' \Rightarrow \delta, \gamma'$ by iterating this construct over γ'

Definition 14 (*action of a substitution*)
If $\sigma : \gamma \Rightarrow \delta$, it has a liberalising action, $\cdot\sigma : \mathbf{Term}(l, d, \gamma) \rightarrow \mathbf{Term}(\mathbf{liberal}, d, \delta)$, extending its action on variables to terms.

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

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

Definition 15 (*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.

$$\theta;\sigma : \gamma_o \Rightarrow \gamma_2 \qquad x(\theta;\sigma) = \Big(x\theta\Big)\,\sigma$$

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

$$\rho;\sigma : \gamma_o \Rightarrow \gamma_2 \qquad x(\rho;\sigma) = \Big(x\rho\Big)\,\sigma$$

Lemma 16 (*action of compositions*)
All four compositions of thinnings and substitions act on terms as the sequence of the component actions.

$$t(\theta;\phi) = \Big(t\theta\Big)\,\phi \quad t(\theta;\sigma) = \Big(t\theta\Big)\,\sigma \quad t(\rho;\phi) = \Big(t\rho\Big)\,\phi \quad t(\rho;\sigma) = \Big(t\rho\Big)\,\sigma$$

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

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

for both thinnings and substitutions. □

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

Definition 17 (*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 18 (*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 $\mathbf{Set}$.

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

Note that we may also concatenate substitutions with the same target scope: if $\sigma : \gamma \Rightarrow \delta$ and $\sigma' : \gamma' \Rightarrow \delta$, then $\sigma, \sigma' : \gamma, \gamma' \Rightarrow \delta$.

5 Judging the judgements and ruling the rules

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*.

5.1 The judgement forms

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 19 (*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. When we have a little more apparatus in place, we shall be able to compute precisely the proof obligation which ensures that a rule justifies the preconditions for its premises and the postcondition for its conclusion, given the preconditions for its conclusion and the postconditions for its premises.

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 20 (*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
$\{\textsc{type}\ T_\gamma\}$	$\textsc{univ}\ T_\gamma \square$	$\{\}$	universe
$\{\textsc{type}\ T_\gamma\}$	$T_\gamma \ni \boxed{t_\gamma}$	$\{\}$	type checking
$\{\}$	$\boxed{e_\gamma} \in S_\gamma$	$\{\textsc{type}\ S_\gamma\}$	type synthesis
$\{\}$	$\boxed{x_\gamma}\ :\ S_\gamma$	$\{\textsc{type}\ S_\gamma\}$	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

We may check that a construction is a valid type, which means that it is reasonable to give as an input to type checking or an output from type synthesis. We may check that something already known to be a type is in fact a *universe*, or type of types. We may check the types of constructions or synthesize the type of computations. We may look up the type of a variable. We may check two known types for equality. We may form a judgement which assigns a valid type to a fresh variable and then demands a judgement in the scope extended with that variable.

A judgement with a subject may be called a *validation* — its purpose is exactly to establish trust in its subject. Type construction, type checking, type synthesis and type lookup are validations. A judgement with no subject may be called a *test* — its purpose is to refine our knowledge of things which are already trusted. The universe and type equality judgements are tests. A context extension is either a validation or a test according to the status of its inner judgement.

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. Let us say that such rules are *locally presentable*. Locally presentable rules may extend the context, assigning a valid type to the new most local variable, but they are not explicit about the global context in which they apply.

5.2 A short farewell to contexts

Free variables exist, so we must have contexts to explain them.

Definition 21 (*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. A rule expressed in terms of global judgements is *globally presented.*

- 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 of 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 globally presented.

Definition 22 (*context extension*)

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

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 globally presented rules for looking up the type of a variable in a context.

Definition 23 (*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, to say nothing of substitution.

We shall develop the technical apparatus to achieve stability with respect to actions on free variables shortly, but for now, let me give the high-level idea. If you do not talk about free variables, you cannot assert anything which can be falsified by an action on free variables.

Dogma 24 (*locally presented rules*)
Only context extension and type lookup are globally presented. Rules for all other judgement forms are locally presented.

Of course, given that variables do occur free in computations, we shall need one rule to synthesize their types. The purpose of the type lookup judgement is to permit its local presentation.

Definition 25 (*variable type synthesis*)

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

However, apart from this rule, access to the global context is absolutely forbidden.

Dogma 26 (*no context access*)
The rules TOP, POP and VAR are the only rules which may use the type lookup judgement, in either premises or conclusion. All other locally presented rules must have ε-judgements for their premises and conclusions, referring only to schematic variables and term variables which are bound locally within judgements, either by abstraction or by context extension.

Intuitively, it is clear that variable lookup is stable under thinning. Inserting more free variables does not preclude access to those we already know about. There is no shadowing in a de Bruijn system. We shall similarly achieve stability under substitution by replacing uses of the VAR rule with other synthesis derivations whose types match, thinned as necessary. In order to ensure that these intuitions can be realised, let us stop talking about free variables before we say something we have cause to regret.

5.3 Change of direction and type equality

The configurable parts of a type theory in this setting are the rules which police constructions, whether they represent types, values or eliminators. Let us treat the remaining syntactic constructs once and for all.

Definition 27 (*change of direction*)
The following rules shall exist in all systems.

$$\textsc{thunk}\ \frac{n \in S \quad S = T}{T \ni [n]} \qquad \textsc{universe}\ \frac{n \in S \quad \textsc{univ}\ S}{\textsc{type}\ [n]} \qquad \textsc{radical}\ \frac{\textsc{type}\ T \quad T \ni t}{(t : T) \in T}$$

For the purposes of this paper, type equality will be given by the axiom

$$\textsc{reflexivity}\ \frac{}{T = T}$$

which is the only nonlinear rule I shall permit here, but it serves to make an insistence on linearity in schematic variables — at least, those which stand for types — unproblematic. The THUNK rule insists that when we have a synthesized type and a type to check, they must agree precisely (or, for humans, they must agree up to the renaming of bound variables). Meanwhile, we may have computations in a type only if the type synthesized for that computation is recognizably a type of types. Lastly, while a radical offers us a candidate for the type, T, to synthesize, we must check both that it really is a type, and, now that T is known to be a type, that T accepts the term, t.

It is worth remarking that more generous notions of type equality are worthy of consideration. Moreover, as the THUNK rule is clearly *directed*, we could relax the requirement to the demand that S be a *subtype* of T. That might be one way to arrange for a Russell-style cumulative hierarchy of universes. Let us leave those possibilities for the future.

We have now dealt with all the generic parts of bidirectional type systems. Before we turn to the machinery which allows us to specify individual type theories by classifying the constructions which occur in them, let us establish a concrete running example from which to generalise.

5.4 A bidirectional reconstruction of Martin-Löf's 1971 type theory

I take Martin-Löf's 1971 type theory (Martin-Löf, 1971) as the exemplary system for two reasons. Firstly, it is small:

Definition 28 (*bidirectional type-in-type*)

$$\frac{}{\textsc{type}\ \star} \qquad \frac{\textsc{type}\ S \quad x\!:\!S \vdash \textsc{type}\ T}{\textsc{type}\ (\Pi\, S\ \backslash x\, T)} \qquad \frac{}{\textsc{univ}\ \star}$$

$$\frac{\textsc{type}\ T}{\star \ni T} \qquad \frac{x\!:\!S \vdash T \ni t}{(\Pi\, S\ \backslash x\, T) \ni \backslash x\, t} \qquad \frac{\bullet \in (\Pi\, S\ \backslash x\, T) \quad S \ni s}{\bullet\, s \in T/(s : S)}$$

Secondly, it is inconsistent: it is famously possible to construct a term which does not β-normalize. We shall not be able to appeal to normalization in our efforts to establish any of its other metatheoretic properties, such as subject reduction.

In addition to $()$, we have two atoms, $\star$ and Π, which tag our canonical types — the universe and function spaces, respectively. We further assert that $\star$ may stand as a type of types, which allows types to be computations for which the type $\star$ may be synthesized. The inconsistency arises from the rule which makes any type, including $\star$ itself, checkably an inhabitant of $\star$. A function type $(\Pi\, S\ \backslash x\, T)$ tells us how to check an abstraction, which we need not tag with a constructor. The $\bullet$ symbol in the synthesis rule for application is a schematic variable: its peculiar name is because of a special property which will shortly become clear. When we can synthesize a function type for the target of an elimination, that tells us to interpret the entire eliminator as the function's argument and check that it inhabits the function's domain.

Of course, this is not quite the whole story. We shall need to equip our theory with the possibility to *compute* in types, with a reduction system involving the β-rule

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

but let us address that requirement later, when we have developed the precise language in which to talk about reductions.

For now, let us focus on checking and synthesizing types. We can see that rules for judgements $\textsc{type}\ T$ and $T \ni t$ trade only in essential constructions, with premises checking components of the conclusion's subject. Meanwhile, the synthesis rule makes no syntactic analysis of its target, requiring only a particular construction for its synthesized type, in order to check the construction of the eliminator and deliver the synthesized type of the whole.

The typing rules do not talk about free variables. The only schematic variable which stands for a computation is the $\bullet$ in the application rule, and it is used in a very particular pattern.

Dogma 29 (*targets of eliminations*)
We are free to write synthesis rules only for eliminations. These rules must name the target $\bullet$ but not analyse its syntactic structure. The first premise of an elimination rule must synthesize the type of $\bullet$.

That is, the only real choices we make, even in the synthesis rules, are how to analyse constructions.

5.5 How rules talk about constructions: patterns and expressions

The formulae which occur in typing rules are not terms of the underlying type theory. They contain *schematic variables* and stand for the sets of terms generated by instantiating those schematic variables. If we are to achieve a generic approach to metatheory, we shall have to be precise about what constitutes these formulae, and how the schematic variables are scoped.

The modes of the places in the judgement forms determine whether a given construction is being analysed by the rule or synthesized by it. Which formulae are appropriate depends on which of these two activities is happening, so we shall have *two* syntaxes of formulae in typing rules — *patterns* for analysis and *expressions* for synthesis. Patterns contain the binding sites for schematic variables; expressions the use sites.

Dogma 30 (*formula syntax by mode*)
The following table summarises the permitted formulae for each mode in premises and conclusions.

	input	**subject**	**output**
premise	expression	schematic variable	pattern
conclusion	pattern	pattern	expression

Definition 31 (*γ-pattern*)
The formal grammar of patterns, $\mathbf{Pat}(\gamma)$, for a given scope is as follows:

$$\begin{array}{rcll} p_\gamma, q_\gamma & ::= & a & \\ & | & (p_\gamma . q_\gamma) & \\ & | & \backslash x\, q_{\gamma,x} & \\ & | & \theta & \text{where } \exists\delta.\, \theta : \delta \sqsubseteq \gamma \end{array}$$

That is, patterns contain only the essential constructions, together with placeholders bearing a thinning whose purpose is to specify which of the variables in γ may occur in the place. They go where constructions go. The same Lisp convention for avoiding dots in right-nested tuples applies. Formally, the pattern used for the function type in our example theory is

$$(\Pi\, \mathbf{1} \,\backslash\, \mathbf{1})$$

with two placeholders, the second of which allows dependency on the bound variable.

It is straightforward to define the action of a thinning on a pattern: if $\theta : \gamma \sqsubseteq \gamma'$ then $\cdot\theta : \mathbf{Pat}(\gamma) \to \mathbf{Pat}(\gamma)'$. The effect of this is to grow the notional scope of the pattern while forbidding any of the 'extra' variables to occur in the pattern's places.

Informally, of course, we write distinct names in the places: these are the binding sites for the schematic variables. We may omit the identity thinning, hence

$$(\Pi\, S \,\backslash x\, T)$$

If we wish a different thinning, we may attach a square bracket to the name, listing the permitted variable. We could also have written:

$$(\Pi\, S[] \,\backslash x\, T[x])$$

Meanwhile, the pattern $(\Pi\, S[] \,\backslash x\, T[])$ matches only function types which happen to be non-dependent.

Such are the patterns for constructions. The only patterns we ever need for computations take the form $\bullet\, p$.

Every pattern has a *domain*, which is a left-nested sequence of scopes: $\varepsilon, \delta_{n-1}, \ldots, \delta_0$. Domains are to schematic variables as scopes are to term variables, and I refer to them by the metavariables ω and χ. All of our machinery related to thinnings works perfectly well for domains, embedding one sequence in other by inserting zero or more new entries. Formally, then, we may select a schematic variable from a domain by giving a singleton thinning, $\xi : \varepsilon, \gamma \sqsubseteq \omega$.

Definition 32 (*domain of a pattern*)
The *domain* of a pattern is computed as follows:

$$\begin{array}{lcl} \mathbf{dom}(a) & = & \varepsilon \\ \mathbf{dom}((p.q)) & = & \mathbf{dom}(p), \mathbf{dom}(q) \\ \mathbf{dom}(\backslash x\, q) & = & \mathbf{dom}(q) \\ \mathbf{dom}(\theta) & = & \mathbf{src}(\theta) \end{array}$$

where ω, ω' is concatenation of domains and $\mathbf{src}(\theta) = \delta$ whenever $\theta : \delta \sqsubseteq \gamma$. We may write $\mathbf{dom}(\vec{p})$ for the concatenation of the domains of a sequence of patterns.

Let us now say what are the expressions, relative to a domain of schematic variables and a scope of term variables.

Definition 33 (*ω-γ-expression*)
The expressions $\mathbf{Exp}(\omega, \gamma)$ are liberal constructions in the full syntax of terms, augmented by the liberal constructions ξ/σ, where $\xi : \varepsilon, \delta \sqsubseteq \omega$ and $\rho : \delta \Rightarrow \gamma$. Expressions in the synthesis rule for an elimination may also contain the essential computation, $\bullet$, standing for the target. These we classify as $\mathbf{Exp}^{\bullet}(\omega, \gamma)$.

Informally, just as we write names at the binding sites of schematic variables in patterns, we refer to them by names instead of some ξ. We may omit the initial ε, of a substitution, and we may omit the $/$ when the substitution, ρ is empty. The binding site of a schematic variable determines its scope, so we need not say which variables are being substituted.

It is straightforward to extend the action of thinnings $\cdot\theta$ and substitutions $\cdot\sigma$ from terms to expressions as we can compute the compositions $\rho;\theta$ and $\rho;\sigma$.

By inspection, our example theory conforms to Dogma 30, using patterns $\star$, $(\Pi\, S\, \backslash x\, T)$, $\backslash t$ and $\bullet\, s$ where only patterns are permitted, and in expression positions, only schematic variables and $T/(s:S)$. We have already excluded the mention of free term variables from our rules, but there is still the matter of which schematic variables we may use where. Let us now be entirely precise.

5.6 What is the domain?

The following dogma specifies the permitted usage of schematic variables in the premises and conclusion of a configurable rule.

Dogma 34 (*clockwise flow*)

Let a rule take the form

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

Let $\vec{p}_i$ be the sequence of patterns in the outputs of J_i and $\vec{p}'$ the sequence of patterns in the *inputs* of J'. Let q be the subject of J' if it has one and $()$ otherwise. We may compute sequences of domains $\omega_1, \ldots, \omega_{n+1}$ of *trusted* schematic variables and $\chi_0, \ldots, \chi_1$ of *untrusted* schematic variables. We shall require the expressions in the inputs of J_i to have domain ω_i and the subject of J_i to be chosen from χ_i. The expressions in the outputs of J' must have domain ω_{n+1}. We require that $\chi_{n+1} = \varepsilon$. The computation proceeds as follows:

- initially, $\omega_1 = \mathbf{dom}(\vec{p}')$ and $\chi_1 = \mathbf{dom}(q)$;
- if J_i has no subject, then $\omega i + 1 = \omega_i, \mathbf{dom}(\vec{p}_i)$ and $\chi_{i+1} = \chi_i$;
- if J_i has context extensions yielding scope γ and a subject, then that subject must be specified by a pair $\xi\,\theta$, such that $\xi : \varepsilon, \delta \sqsubseteq \chi_i$ and $\theta : \delta \sqsubseteq \gamma$; that being the case, $\omega_{i+1} = \omega_i, \mathbf{dom}(\vec{p}_i), \delta$ and $\chi_{i+1} = \chi_i - \xi$, i.e., χ_i with ξ removed.

What is going on? The *trusted* ω_i grow as we work our way clockwise around the rule, starting with the schematic variables which come from the conclusion's inputs: we trust them in the sense that we know the precondition which must hold for them. The *untrusted* χ_i come from the subject's conclusion and shrink to nothing as we work our way along the premises. Each premise grows the trusted domain by the schematic variables of its outputs (trusted because of postconditions) and moves its subject, if it has one, out of the untrusted domain and into the trusted: the premise states the trust which has been established. The thinning in a premise subject ensures that the term variables in scope for that schematic variable map to term variables locally bound in the context: informally, we achieve this by artful name capture, as in the premise $x{:}S \vdash T \ni t$ of the rule for $\backslash x\, t$.

Let me emphasize a key consequence of Dogma 34: subjects of premises come *only* from the subject of a conclusion; a rule establishes trust in *every* part of its conclusion's subject; trust, once established, is *never* revisited.

We may readily check that our example rules all conform to Dogma 34.

5.7 How to apply a rule

Dogma 34 makes it intuitive that rules are applied to actual terms by matching patterns to compute environments and instantiating expressions with environments. Let us make this intuition precise.

Definition 35 (*ω-environment*)
A given domain ω determines a notion of environment, $\mathbf{Env}(\omega)$. We may refer to such environments by metavariable Ω, and define them inductively as follows:

$$\frac{}{\varepsilon : \mathbf{Env}(\varepsilon)} \qquad \frac{\Omega : \mathbf{Env}(\omega) \quad t : \mathbf{Term}(\mathbf{liberal}, \mathbf{construction}, \delta)}{\Omega, t : \mathbf{Env}(\omega, \delta)}$$

Again, we may use a comma to concatenate whole environments, as well as to extend them with a single term.

Pattern matching rests on dependency checking, known colloquially as *thickening*, as it is the partial inverse of thinning.

Definition 36 (*thickening*)
If $\theta : \delta \sqsubseteq \gamma$ then $\theta\cdot : \mathbf{Term}(l,d,\gamma) \rightharpoonup \mathbf{Term}(l,d,\delta)$, specified by the assertion

$$\theta t = s \;\Leftrightarrow\; t = s\theta$$

We may compute θt by a failure-propagating structural recursion on t. It suffices to consider the cases for abstraction and variables, as all others proceed componentwise. Straightforwardly, $\theta\left(\backslash x t\right) = \backslash x\left((\theta 1)t\right)$. For variables, recall that the variable we write as x is really a singleton thinning $\phi : \varepsilon, x \sqsubseteq \gamma$: let $(\theta', \phi') = \mathbf{pullback}(\theta, \phi)$ and return θ' if $\phi' = \mathbf{1}$, as by the definition of a pullback, we have $\theta'; \theta = \mathbf{1}; \phi = \phi$.

We may thus characterize pattern matching as a partial function yielding environments.

Definition 37 (*pattern matching*)

$$\mathbf{match}(\cdot,\cdot) : (p : \mathbf{Pat}(\gamma)) \times \mathbf{Term}(\mathbf{liberal}, \mathbf{construction}, \gamma) \rightharpoonup \mathbf{Env}(\mathbf{dom}(p))$$

$$\begin{array}{ll}
\mathbf{match}(a,a) & = \varepsilon \\
\mathbf{match}((p.q),(s.t)) & = \mathbf{match}(p,s), \mathbf{match}(q,t) \\
\mathbf{match}(\backslash x q, \backslash x t) & = \mathbf{match}(q,t) \\
\mathbf{match}(\theta,t) & = \varepsilon, \theta t
\end{array}$$

Remember, though, that the patterns in our locally presented rules scope only over the *locally* bound variables, by Dogma 26. We shall need to match terms which contain the free variables that the rules are not allowed to talk about.

Definition 38 (*shifting of domains and patterns*)
Let γ, ω be the domain obtained by mapping every scope δ in ω to the concatenation γ, δ. Let γ, p be the pattern obtained by mapping every placeholder θ in p to the concatenation $\mathbf{1}_\gamma, \theta$. For the latter, we obtain

$$\gamma, \cdot : \mathbf{Pat}(\gamma') \to \mathbf{Pat}(\gamma, \gamma')$$

Lemma 39 (*coherence of shifting*)

$$\mathbf{dom}(\gamma, p) = \gamma, \mathbf{dom}(p)$$

Proof
Proceed by structural induction on p, using the fact that mapping commutes with concatenation. □

If γ is the scope of free variables and γ' the scope of variables locally bound in a rule, we may take a pattern $p : \mathbf{Pat}(\gamma')$ and a term $t : \mathbf{Term}(\mathbf{liberal}, \mathbf{construction}, \gamma, \gamma')$, and compute $\mathbf{match}(\gamma, p, t)$, obtaining, if we are lucky, an $\mathbf{Env}(\gamma, \mathbf{dom}(p))$.

What happens to expressions? We should be able to instantiate an expression $t : \mathbf{Exp}(\omega, \gamma)$ with an environment $\Omega : \mathbf{Env}(\gamma, \omega)$, and obtain a $t\Omega : \mathbf{Term}(\mathbf{liberal}, \mathbf{construction}, \gamma)$. We shall need a little room to manoeuvre before we begin.

Lemma 40
functoriality of environments We may extend $\mathbf{Env}(\cdot, \omega)$ to functors from thinnings or substitutions to $\mathbf{Set}$.

Proof

Suppose $\Omega : \mathbf{Env}(\gamma,\omega)$, $\theta : \gamma \sqsubseteq \gamma'$ and $\sigma : \gamma \Rightarrow \gamma'$. We define $\Omega\theta$ and $\Omega\sigma$ to act pointwise on the terms t in Ω over γ,δ for each δ in ω. The image of t under $\cdot\theta$ is $t(\theta\delta)$, while for $\cdot\sigma$, we obtain $t(\sigma\delta)$. The functor laws hold because terms are also functors from thinnings and substitutions, and the iterated weakening $\cdot\delta$ is an endofunctor on both thinnings and substitutions. □

With this in place, we may now push environments under binders.

Definition 41 (*expression instantiation*)

$$\frac{t : \mathbf{Exp}(\omega,\gamma) \quad \Omega : \mathbf{Env}(\gamma,\omega)}{t\Omega : \mathbf{Term}(\mathbf{liberal},\mathbf{construction},\gamma)}$$

The operation proceeds by structural recursion on t. We propagate Ω componentwise, except when we encounter an abstraction or a schematic variable:

- $\left(\backslash x\, t\right)\Omega = \backslash x\left(t\hat{\Omega}\right)$
- $\left(\xi/\rho\right)\Omega = t(\iota_\gamma,\rho)$ where $\xi : \varepsilon,\delta \sqsubseteq \omega$, thence $t : \mathbf{Term}(\mathbf{liberal},\mathbf{construction},(\gamma,\delta))$ is the projection at ξ from Ω.

⋆⋆⋆

Definition 42 (*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 43 (*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 44 (*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 26 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. □

We are nearly ready to consider how, in general, to construct the typing rules for particular theories, but we have three more rules to give which must be present in any theory, characterising the relationship between types, type checking and type synthesis.

These rules all conform to 26.

The seasoned type theorist will further notice the total absence from the rules so far of any form of *computation*. Our type systems are, thus far, entirely *inert*. That much I shall remedy after we have introduced the means of creating a redex, but for that, we shall need to check canonical constructions and synthesize types for computations. And in order to do that, we should think further about which typing rules make sense by construction.

6 Ruling on rules

in which I introduce the formal notions of 'pattern' and 'expression', with a systematic treatment of the schematic variables in rules; I also give the permitted shapes of rules which check constructions and rules which synthesize types for eliminations; I give the scoping dogma for schematic variables in rules, in particular the dogma which insists that subjects in premises must come from the subjects of the conclusion; I demonstrate stability under substitution

7 Questions better left unasked

in which I investigate further the consequences of the deliberately restrictive language of patterns; the structural rules for reduction ensure that pattern matching is stable with respect to reduction; patterns have simple unification, ensuring the appropriate invertibility of rules; on syntax-directedness and its consequences; I specify the sanity clause and prove its decidability

8 Mind diamonds!

in which I show how a system of syntax-directed typing rules demands a set of β-rules which are non-overlapping by construction, resulting in the definability of a Takahashi-style notion of 'development', and thence confluence — parallel reduction has the diamond property by construction

In our reconstruction of Martin-Löf's 1971 theory, the demanded β-rule is

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

For best results, take $? = t/(s : S)$.

9 Subject reduction by construction

in which I demand that the β-rules be well typed in the inert fragment of a type theory, and show that compliance with this demand is decidable; I then prove that confluence is sufficient to ensure that subject reduction holds in the presence of the following

$$\text{PRE}\ \frac{T \rightsquigarrow T' \quad T' \ni t}{T \ni t} \qquad \text{POST}\ \frac{e \in S \quad S \rightsquigarrow S'}{e \in S}$$

In our reconstruction of Martin-Löf's 1971 theory, inertly inverting the typing rules for the left-hand side yields

$$\textsc{type}\ S \quad x{:}S \vdash \textsc{type}\ T \quad S \ni s \quad x{:}S \vdash T \ni t$$

from which we may inertly deduce that

$$T/(s : S) \ni t/(s : S)$$

and hence that Martin-Löf's 1971 theory has the subject reduction property.

10 Discussion

in which I review related work and progress, then set out the prospectus for the research programme opened by this paper

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.