Under consideration for publication in J. Functional Programming

The types who say 'ni'

CONOR 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 dependent type systems, where the usual type *synthesis* judgement, $t : T$, is partly replaced (specifically for introduction forms) by the *checking* judgement, $T \ni t$, while elimination forms continue to have their types extracted, $e \in S$. Bidirectional typing (or 'local type inference') has been with us for some time (Pierce & Turner, 2000). It has been something of a secret weapon for implementing dependent type systems (Asperti *et al.*, 2012): it is overdue to put the metatheory on a solid footing. The literature offers diverse conventions for labelling colons with arrows, but I exploit the asymmetry of $\ni$ and $\in$ (`\ni` and `\in` in LaTeX, respectively) to ensure the left-to-right progress of *time*. The types who say 'ni' come before, and they know the terms they will accept.

I illustrate the approach 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 the former 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 some 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	$\dfrac{x{:}S \vdash t \ : \ T}{\lambda x.t \ : \ (x{:}S) \to T}$	$\dfrac{\mathrm{TYPE}\ S \quad x{:}S \vdash t \ : \ T}{\lambda x{:}S.t \ : \ (x{:}S) \to T}$
pairs	$\dfrac{s \ : \ S \quad t \ : \ T[s/x]}{(s,t) \ : \ (x{:}S) \times T}$	$\dfrac{s \ : \ S \quad x{:}S \vdash \mathrm{TYPE}\ T \quad t \ : \ T[s/x]}{(s,t)_{x.T} \ : \ (x{:}S) \times T}$

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 every time.

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. Moreover, once we can compute types from data, there is no reason to believe that the second component is even a length-indexed list at all, when the first is any number other than 3.

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.

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

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

$$\frac{\Gamma \vdash t\ :\ T \quad t \rightsquigarrow t'}{\Gamma \vdash t'\ :\ T} \qquad \frac{\Gamma \vdash \mathrm{TYPE}\ T \quad T \rightsquigarrow T'}{\Gamma \vdash \mathrm{TYPE}\ T'}$$

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

$$\textsc{type}\ (x{:}S) \to T \quad (x{:}S) \to T \rightsquigarrow (x{:}S') \to T$$

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, avoid 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 has 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.

In what follows, I shall adopt a mathematical style, identifying and numbering specific Definitions, Lemmas and Theorems, but introducing a new keyword — *Dogma*. A dogma is a *religious* truth, rather than a *Platonic* truth: it is made true by human endeavour, rather than by nature. A dogma is a design choice — a prejudice, if you like. However, the prejudice should be judicial: every dogma should be motivated by some anticipated benefit. Moreover, every numbered Dogma will have a 'Proof', its exegesis, in the form of a forward pointer to the Definition that ensures the Dogma is followed. Let us begin.

Dogma 1 (*generic syntax*)

One syntax is sufficient to encode the types and terms of all the systems that we shall seek to explain. It is not expected that being syntactically well formed is the same thing as being meaningful. It is enough to explain what it is to be a classifier, i.e., a *type*, and what it is to be classified.

Proof

The generic syntax is given in Definition 2. The means to manage classification — the notion of typing judgement — is given in Definition 20. □

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 2 (*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
const*ruction*	k_γ, K_γ $::= \quad a$ $\mid \quad (s_\gamma . t_\gamma)$ $\mid \quad \backslash x\, t_{\gamma,x}$	$s_\gamma, t_\gamma, S_\gamma, T_\gamma$ $::= \quad k_\gamma$ $\mid \quad \underline{n_\gamma}$
computation	n_γ, N_γ $::= \quad x_\gamma$ $\mid \quad e_\gamma\, s_\gamma$	$e_\gamma, f_\gamma, E_\gamma, F_\gamma$ $::= \quad n_\gamma$ $\mid \quad (t_\gamma : T_\gamma)$

Let us write $\mathsf{Term}\, l\, d\, \gamma$ for the set of terms in the quadrant given by *liberality* l and *direction* 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. I use teletype parentheses, `(...)` as 'LISP brackets' to leave $(\ldots)$ free for their usual mathematical purpose of resolving ambiguity. Let us further adopt the LISP convention that `.(`*stuff*`)` may be abbreviated as *stuff*, 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. Formally, working de Bruijn style, there is no reason to name the variable bound by an abstraction, but humans seem to appreciate it.

The *liberal* constructions extend the canonical constructions with *thunks*, $\underline{n}$ — 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* computations 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. Radicals are permitted only in eliminations, and these are the only eliminations which compute. That is...

Dogma 3 (*typed computation*)
There is no computation step which proceeds uninformed by the type at which computation is happening.

Proof
Definition 28 will ensure that reduction is the no more than the contextual closure of whatever may happen to eliminated radicals. □

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.

$$\underline{(t : T)} = t$$

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, and ensure that it is always clear from context which variable in t is being substituted — the forward slash of substitution answers the backslash of abstraction.

Example 4 (*Martin-Löf's 1971 theory: syntax and β-reduction*)
Let Π and $\star$ be atoms. The canonical types are given as $\star$ and $(\Pi\, S\, \backslash x\, T)$, with the former being the type of all types and the latter being the type of abstractions, $\backslash x\, t$. The β-rule is

$$(\backslash x\, t : (\Pi\, S\, \backslash x\, T))\; s \rightsquigarrow_\beta (t/(s : S) : T/(s : S))$$

Note that the β-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, in general.

Dogma 5 (*β-rules*)
All β-rules have form

$$(t : T)\ s \rightsquigarrow_\beta\ (t' : T')$$

Further, each of t, T and s may demand only essential construction structure.

Proof
This is the substance of Definition 28. □

Were we to add pair types, with atoms `car` and `cdr` as the eliminators, we could have:

$$((s\,.\,t) : (\Sigma\, S \setminus x\, T))\ \texttt{car} \rightsquigarrow_\beta\ (s : S) \qquad ((s\,.\,t) : (\Sigma\, S \setminus x\, T))\ \texttt{cdr} \rightsquigarrow_\beta\ (t : T\,/\,(s : S))$$

By adopting this uniform presentation of computation, we have already taken a major step towards establishing *confluence* of β-reduction: if one β-redex contains another, the inner redex will always be copied zero or more times whenever the outer redex contracts. Critical pairs can arise only if the same term is a redex for two different β-rules.

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.

Remark 6 (*run-time type erasure*)
The fact that computation is formally conducted in the presence of types is motivated by clarity of metatheory, rather than by pragmatic considerations of efficient closed-term execution. That said, one can achieve a sensible notion of type erasure by ensuring that the target term and eliminator of an eliminated radical determine the reduction rule applicable, and that type information therein is used only to determine type information in the reduct. By inspection, this is the case for our examples.

4.2 Thinnings and substitutions

To keep a tight grip in our formal account of *scope*, let us develop the apparatus which explains when one scope embeds in another.

Definition 7 (*thinning*)
A thinning is an order preserving embedding between scopes

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

I typically use θ and ϕ 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, in effect, 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.

Definition 8 (*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 9 (*composition of thinnings*)
If $\theta : \gamma_0 \sqsubseteq \gamma_1$ and $\phi : \gamma_1 \sqsubseteq \gamma_2$, then $\theta\phi : \gamma_0 \sqsubseteq \gamma_2$ defined as follows:

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

Note that I adopt the *diagrammatic* convention for composition, written just as juxtaposition. When we see $\theta\phi$, we should remember that they meet in the middle, with θ's source on the left and ϕ's target on the right.

You will not be surprised to learn that thinnings have categorical structure, to be established formally in section 7.1. For the moment, though, let us establish the minimum machinery necessary to proceed with our analysis of type systems.

The formal truth is that a 'variable' x_γ is a thinning in some $x \sqsubseteq \gamma$, i.e., from the singleton scope to γ. Noting that γ occurs positively in the grammar, we obtain an action of thinnings on our syntax.

Definition 10 (*action of a thinning*)
If $\theta : \gamma \sqsubseteq \delta$, it has a covariant, quadrant-preserving action, $\cdot\theta : \mathsf{Term}\, l\, d\, \gamma \to \mathsf{Term}\, l\, d\, \delta$, which reduces to composition at variables.

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

I write the action postfix as a reminder that the scope of $t\theta$ is the target scope of θ, i.e., that the action is indeed covariant.

By contrast, thinnings act *contravariantly* on vectors of any kind. Viewed left-to-right, thinnings represent the *injection* of one scope into a larger scope. Viewed right-to-left, thinnings represent the *selection* of a subset.

Definition 11 (*selection*)
If X is a set, let X^δ be the set of vectors with one component in X for each variable in δ. If $\theta : \gamma \sqsubseteq \delta$ and $\vec{x} : X^\gamma$, then let $\theta!\vec{x} : X^\gamma$ be defined thus:

$$\varepsilon!\varepsilon = \varepsilon \qquad (\theta, 0)!(\vec{x}, x) = \theta!\vec{x} \qquad (\theta, 1)!(\vec{x}, x) = (\theta!\vec{x}), x$$

I write the $\theta!\cdot$ action prefix as a reminder that the length of $\theta!\vec{x}$ is given by the source scope of θ. In due course, we shall have that selection is functorial, and natural in the type of elements.

We shall have need of left and right injections.

Definition 12 (*injections to a concatenation*)

We take

$$\gamma \triangleleft \delta = \mathbf{1}_\gamma, \mathbf{0}_\delta : \gamma \sqsubseteq \gamma, \delta \qquad \gamma \triangleright \delta = \mathbf{0}_\gamma, \mathbf{1}_\delta : \delta \sqsubseteq \gamma, \delta$$

Next, let us have *substitutions*, which act simultaneously on all variables in a scope. Let us take σ and ρ as metavariables for substitutions.

Definition 13 (*substitution*)
Let $\gamma \Rightarrow \delta$ be the set of substitutions from γ variables to δ terms, i.e., vectors

$$\sigma : (\mathsf{Term}\,\mathsf{lib}\,\mathsf{comp}\,\delta)^\gamma$$

Whenever $\theta : \gamma' \sqsubseteq \gamma$, we have $\theta!\sigma$ for the substitution in $\gamma' \Rightarrow \delta$ given by selecting from σ only the images corresponding to variables chosen by θ. In particular, $x!\sigma$ is the image of some variable (i.e., singleton thinning) x. We may write ε for the trivial substitution from the empty scope. If $\sigma : \gamma \Rightarrow \delta$ and $e : \mathsf{Term}\,\mathsf{lib}\,\mathsf{comp}\,\delta$, then we have $\sigma, e : \gamma, x$.

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 14 (*weakening*)
The thinning $\mathbf{1}, 0 : \gamma \sqsubseteq \gamma, x$ 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 $\theta \uparrow$.

Definition 15 (*action of a thinning on a substitution*)
If $\sigma : \gamma \Rightarrow \delta$ and $\theta : \delta \sqsubseteq \delta'$, we write $\sigma\theta : \gamma \Rightarrow \delta'$ for the pointwise action of θ on the computations in σ. In particular,

$$\hat{\sigma} = \sigma \uparrow : \gamma \Rightarrow \delta, x$$

We now have all we need to make substitutions act on our syntax.

Definition 16 (*action of a substitution*)
If $\sigma : \gamma \Rightarrow \delta$, it has a liberalising action, $\cdot\sigma : \mathsf{Term}\, l\, d\, \gamma \to \mathsf{Term}\,\mathsf{lib}\, d\, \delta$, which is projection at variables.

$$\begin{array}{rcl@{\qquad}rcl} a\sigma & = & a & \underline{n}\sigma & = & \underline{n\sigma} \\ (s\,.\,t)\,\sigma & = & (s\sigma\,.\,t\sigma) & & & \\ (\backslash x\, t)\sigma & = & \backslash x\, t(\hat{\sigma}, x) & & & \end{array}$$

$$\begin{array}{rcl@{\qquad}rcl} x\sigma & = & x!\sigma & (t : T)\,\sigma & = & (t\sigma : T\sigma) \\ (e\, s)\sigma & = & e\sigma\, s\sigma & & & \end{array}$$

Note that if $x!\sigma = (t : T)$, then $\underline{x}\sigma = \underline{(t : T)} = t$, as the smart $\underline{\cdot}$ strips the type annotation.

Substitutions also have categorical structure, as we shall see in section 7.3. For now, let us establish the operations.

Definition 17 (*identity substitution*)
The identity substitution $\iota_\gamma : \gamma \Rightarrow \gamma$ is given by

$$\iota_\varepsilon = \varepsilon \qquad \iota_{\gamma, x} = \hat{\iota}_\gamma, x$$

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

Definition 18 (*composition of substitutions*)
If $\rho : \gamma_0 \Rightarrow \gamma_1$ and $\sigma : \gamma_1 \Rightarrow \gamma_2$, their composition, $\rho\sigma$ makes σ act *pointwise* on terms in ρ.

$$\rho\sigma : \gamma_o \Rightarrow \gamma_2 \qquad x!(\rho\sigma) = (x!\rho)\sigma$$

We have established the *definitions* of thinnings and substitutions, with their actions on terms. Let us acknowledge the debt outstanding to establish their *structure*, but resist the temptation to dive into detail now. Let us instead zoom out and develop more of the ideas behind bidirectional type systems.

5 Judging the judgements

Typing *judgements* make statements about the terms of our theories. The acceptable *derivations* of judgements are defined inductively, by means of *rules*, each of which has zero or more *premises* and one *conclusion* — *formal judgements* which give rise to actual judgements by instantiation of their *schematic variables*.

We should be clear about the distinction between the actual judgements derived in the process of checking the actual types of actual terms, and the formal judgements which we write when we specify the typing rules. We should hope to gain useful results about the derivability of actual judgements by taking care with the language we use for formal judgements. This section's primary concern is with actual judgements, but we should hope to learn some lessons informing the treatment of formal judgements in the next section.

5.1 The judgement forms

The judgement forms are specified as sequences of *punctuation* and *places*. An *actual* judgement has terms in its places. A formal judgement has *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. Actual judgements have some scope, γ, and I shall be careful to give scopes to their places, determining the scopes of the terms we may put into them. 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 postconditions in braces to the right.

Definition 20 (γ*-judgement*)
The judgements, J_γ, in scope γ are given inductively as follows:

$\mathsf{Pre}\, J_\gamma$	J_γ	$\mathsf{Post}\, J_\gamma$	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
$\{\textsc{type}\ S_\gamma, x{:}S_\gamma \vdash \mathsf{Pre}\, J'_{\gamma,x}\}$	$x{:}S_\gamma \vdash J'_{\gamma,x}$	$\{x{:}S_\gamma \vdash \mathsf{Post}\, J'_{\gamma,x}\}$	context extension

We may extend the postfix action of thinnings and substitutions to judgements, writing $J\theta$ and $J\sigma$, respectively.

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 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 Contexts: hello, and goodbye

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

Definition 21 (*γ-context, actions on contexts, global judgements, context validity*)
A γ-context, Γ is a vector, mapping variables in γ to liberal constructions over γ:

$$\Gamma : (\mathsf{Term}\,\mathsf{lib}\,\mathsf{cons}\,\gamma)^{\gamma}$$

Informally, we write it as a sequence of type assignments, $x_1 : S_1, \ldots, x_n : S_n$. We may write $\theta!\Gamma$ for selections from Γ, thence $x!\Gamma$ for projections from Γ. Meanwhile, thinnings and substitutions act pointwise as $\Gamma\theta$ and $\Gamma\sigma$ respectively. In particular, $\hat{\Gamma} = \Gamma\uparrow$. Given such a Γ, 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*. Context *validity* is defined inductively, as follows:

- ε is a valid ε-context.
- If Γ is a valid γ-context and $\Gamma \vdash \textsc{type}\ S$, then $\hat{\Gamma}, x : \hat{S}$ is a valid γ, x-context.

At least two things may strike you as peculiar.

- The property that the types of later variables may depend only on earlier variables is not encoded in the *syntax* of contexts, but only in their *validity*. The consequence of this nonstandard choice is that the selective action of thinning makes syntactic sense, and in particular, the projection $x\Gamma$ from a γ-context yields a term in γ, which is exactly what we need when we *use* the context.
- 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 are but two globally presented rules:

Definition 22 (*context extension, type lookup*)

$$\textsc{extend}\ \frac{\hat{\Gamma}, x : \hat{S} \vdash J}{\Gamma \vdash x : S \vdash J} \qquad \textsc{lookup}\ \frac{}{\Gamma \vdash x \in x!\Gamma}$$

The client invoking the EXTEND 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, the client invoking the LOOKUP rule must again ensure that Γ is valid, which will allow the server to guarantee that $\Gamma \vdash \textsc{type}\ x!\Gamma$, provided we can ensure that type construction is stable under thinning.

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 never talk about free variables, the truths you tell will be preserved by operations which act only on free variables.

Dogma 23 (*locally presented rules*)
Only context extension and type lookup are globally presented. Rules for all other judgement forms are locally presented.

Proof
This is borne out in Definitions 24, 26, 27, and 47 through 50. □

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 24 (*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 25 (*no context access*)
The VAR rule is the only rule which may use the type lookup judgement. All other rules must be locally presented, referring only to schematic variables and term variables which are bound locally within judgements, either by abstraction or by context extension.

Proof
This will be a consequence of the formulae permitted in rules by Definitions 45, 46, and 47 through 50. □

When we examine more formally the formulae which may appear in typing rules, let us do so in accordance with this dogma.

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, type equality and reduction

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 26 (*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 not yet treated the possibility of *reduction* in types, without which our notion of type equality is overly restrictive. Calculemus!

Definition 27
type reduction, pre-checking and post-synthesis

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

The above rules allow us to compute a type to a canonical form before checking a construction, or after the synthesis of a target type, so that we can check an elimination. Moreover, either side of the THUNK rule, which insists on exact type equality, we may compute synthesized and checked types to a common reduct.

Reduction is generated by the contextual closure of the β-rules. It is not a judgement, but merely a scope-respecting binary relation on liberal constructions and computations, needing no context. Even so, it should be given a precondition-postcondition specification, making clear which judgements are preserved by reduction: that will be the *subject reduction* property, and we shall attend to it in due course. In the meantime, let us establish what is common to all the reduction systems under this setup.

Definition 28 (*contextual closure of reduction*)

$$\frac{(t:T)\ s \rightsquigarrow_\beta (t':T')}{(t:T)\ s \rightsquigarrow (t':T')}$$

$$\frac{s \rightsquigarrow s'}{(s\,.\,t) \rightsquigarrow (s'\,.\,t)} \qquad \frac{t \rightsquigarrow t'}{(s\,.\,t) \rightsquigarrow (s\,.\,t')} \qquad \frac{t \rightsquigarrow t'}{\backslash x\,t \rightsquigarrow \backslash x\,t'} \qquad \frac{n \rightsquigarrow e}{\underline{n} \rightsquigarrow \underline{e}}$$

$$\frac{e \rightsquigarrow e'}{e\ s \rightsquigarrow e'\ s} \qquad \frac{s \rightsquigarrow s'}{e\ s \rightsquigarrow e\ s'} \qquad \frac{t \rightsquigarrow t'}{(t:T) \rightsquigarrow (t':T)} \qquad \frac{T \rightsquigarrow T'}{(t:T) \rightsquigarrow (t:T')}$$

We may clearly extend reduction pointwise to contexts, $\Gamma \rightsquigarrow \Gamma'$. Let $\rightsquigarrow^*$ be the reflexive transitive closure of $\rightsquigarrow$. Let $\rightsquigarrow^?$ be the union of $\rightsquigarrow$ with equality, i.e., the relation that characterizes taking at most one step.

The fact that these rules offer no base case should not cause alarm: it is for individual systems to identify specific radical eliminations $(t:T)\ s$ as β-redexes. More particularly, if every well typed radical elimination is a redex in at least one way, we shall guarantee

progress, and if every well typed radical elimination is a redex in at most one way, we shall guarantee *confluence*. Subject reduction amounts to showing that the assumptions which must hold for a redex to be well typed are sufficient to show that the redex has the same type. If we are systematic about how to frame these requirements in terms of the typing rules, we shall have these properties by construction.

We now have all the *generic* rules for bidirectional type systems, in the sense of this paper. 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 29 (*bidirectional type-in-type*)

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

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

$$\overline{(\backslash x\, t : (\textstyle\prod S\ \backslash x\, T))\ s \rightsquigarrow_\beta (t/(s{:}S) : T/(s{:}S))}$$

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 $\prod$, 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 $(\prod S\ \backslash x\, T)$ tells us how to check an abstraction, which we need not tag with a constructor. 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.

Let us look more closely at the rules for 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 e in the application rule, and it is used in a very particular pattern.

Dogma 30 (*targets of eliminations*)

We are free to write synthesis rules only for eliminations. These rules must name the target but not analyse its syntactic structure. The first premise of an elimination rule must synthesize the type of the target.

Proof
This will be the substance of Definition 49. □

That is, the only real choices we make, even in the synthesis rules, are how to analyse constructions. It is in the light of that observation that we should ask 'What are the redexes?'. The only elimination rule demands the type $(\Pi\, s\ \backslash x\, T)$ for its target, e. Naïvely ignoring pre- and post-computation for the moment, the only way e can be a radical is if $(\Pi\, s\ \backslash x\, T)$ is a type, and if the canonical term thus annotated is checked at that type, which means it must be $\backslash x\, t$. Working backwards from the type formation, we learn what must be true about S and T; working backwards from type checking, we learn what must be true of t; working backwards from the elimination rule, we learn what must be true of s, and what must be the type component of the radical reduct. We must have

$$\text{TYPE}\ S \qquad x{:}S \vdash \text{TYPE}\ T \qquad x{:}S \vdash T \ni t \qquad S \ni s$$

and we must deliver exactly one β-rule, of the form

$$(\backslash x t : (\Pi\, S\ \backslash x\, T))\ s \rightsquigarrow_\beta (\ ?\ : T/(s{:}S))$$

Our assumptions certainly justify

$$(s{:}S) \in S \qquad x{:}S \vdash (t{:}T) \in T$$

so that if we can ensure stability under substitution, we shall have

$$(t/(s{:}S) : T/(s{:}S)) \in T/(s{:}S)$$

which rather suggests that we take $? = t/(s{:}S)$. Showing that $\text{TYPE}\ T/(s{:}S)$ is necessary, even to discharge the postcondition for the elimination rule. The obligation that is peculiar to the β-rule is to complete the radical by showing $T/(s{:}S) \ni t/(s{:}S)$.

It is instructive to consider what would happen, were we to extend our theory with dependent pairs. We should add

$$\frac{\text{TYPE}\ S \quad x{:}S \vdash \text{TYPE}\ T}{\text{TYPE}\ (\Sigma\, S\ \backslash x\, T)} \qquad \frac{S \ni s \quad T/(s{:}S) \ni t}{(\Sigma\, S\ \backslash x\, T) \ni (s\,.\,t)}$$

$$\frac{e \in (\Sigma\, S\ \backslash x\, T)}{e\ \texttt{car} \in S} \qquad \frac{e \in (\Sigma\, S\ \backslash x\, T)}{e\ \texttt{cdr} \in T/e\ \texttt{car}}$$

Take note that the type of the second projection, e `cdr`, depends on the value of the first, e `car`. That is enough to show that we shall need schematic variables for at least one computation — the target of an elimination — even though our other schematic variables all stand for constructions. That the type of a function application does not depend on the function is a lucky break which we should not take for granted!

Now, for the β-rules, we may play the same game of 'working backwards' to find the assumptions underpinning the existence of redexes. What is that game, exactly? We are *unifying* the type checked in an introduction rule with the target's type in an *elimination*

rule, in order to obtain *all* possible radical eliminations. In this case, we find

$$\begin{array}{lll} ((s\,.\,t)\,{:}\,(\Sigma\, S\setminus x\, T))\,\mathtt{car} & \rightsquigarrow_\beta & (\ ?_{\mathtt{car}}\ {:}S) \\ ((s\,.\,t)\,{:}\,(\Sigma\, S\setminus x\, T))\,\mathtt{cdr} & \rightsquigarrow_\beta & (\ ?_{\mathtt{cdr}}\ {:}T/((s\,.\,t)\,{:}\,(\Sigma\, S\setminus x\, T))\,\mathtt{car}) \end{array}$$

How do we know we have found them *all*?

Dogma 31 (*unification*)
We must ensure that the language for writing types checked in introduction rules and for target types analysed in elimination rules has not only decidable unification, but also *most general* unifiers.

Proof
To be discharged. □

Returning to the problem of finding our reducts, we know

$$\textsc{type}\ S \qquad x{:}S \vdash \textsc{type}\ T \qquad S \ni s \qquad T/(s{:}S) \ni t$$

so we may certainly take $?_{\mathtt{car}} = s$. But may we take $?_{\mathtt{cdr}} = t$, as one might hope? In order to do so, we must immediately precompute with the β-rule for *e car*, yielding

$$\frac{T/((s\,.\,t)\,{:}\,(\Sigma\, S\setminus x\, T))\,\mathtt{car} \rightsquigarrow_\beta T/(s{:}S) \quad T/(s{:}S) \ni t}{T/((s\,.\,t)\,{:}\,(\Sigma\, S\setminus x\, T))\,\mathtt{car} \ni t}$$

That is to say, the justification of some β-rules depends on the deployment of others. We now have a balance to strike, if we want a machine to check our systems of rules: on the one hand, reduction may be nonterminating; on the other, some reduction is necessary. How much reduction is judicious? Let us ponder that anon.

6 Ruling the rules

We have so far been thinking about the judgement forms for our type systems and the actual derivations we can make of actual judgements about actual terms. In doing so, we have identified some informal disciplines to which we should like the typing rules to adhere. It is time to make these disciplines precise and enforce them when we write the formal judgements that comprise the premises and conclusion of each of the typing rules under our control.

6.1 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 32 (*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

Moreover, the type given in a context extension for a premise is an expression.

Proof
The characterization of premises is given in Definitions 45 and 46. The characterization of conclusions is given in Definitions 47 through 50. □

Informally, we have already learned quite a lot about the difference between patterns and expressions. For example, we know that substitution in schematic variables, such as our commonly occurring $T/(s:S)$, makes sense only in *expressions* because we can compute substitutions but not invert them. You may readily check that the substitutions used in our rules so far occur only in expression positions. Moreover, we have learned that the language of patterns must be sufficently weak as to sustain a computable notion of most general unifier: we can have no such hope for expressions.

The scoping of schematic variables in rules will be governed by the following dogma.

Dogma 33 (*clockwise flow*)
Scoping of schematic variables in typing rules flows clockwise around each typing rule, from the inputs and subjects of the conclusion, left-to-right through the premises, then into the outputs of the conclusion. The schematic variables in the outputs of a premise are brought into scope after that premise.

Proof
This will hold by construction as a consequence of Definitions 45, 46, and 47 through 50. □

The effect of this dogma is to ensure a viable strategy for implementing a type checking algorithm. It is clear how we will know each of the things referred to by schematic variables — by pattern matching — in time to make use of them in expressions.

However, there is a further refinement of schematic scoping which has a powerful and subtle impact.

Dogma 34 (*validation of subject components*)
Each schematic variable in a conclusion's subject will be a premise subject exactly once, and no schematic variable may be the subject of a premise unless it is bound by the subject of the conclusion. No subject schematic variable will be used in an expression until after it has been validated as the subject of a premise.

Proof
This will hold by construction as a consequence of Definitions 45 and 46. □

This dogma captures the idea that the schematic variables in the conclusion inputs and the premise outputs are already sufficiently trusted that there is never any need to revalidate

them. The schematic variables in conclusion subjects, however, must be validated before it is safe to use them. We thus achieve a significant decrease in the complexity of subject reduction proofs, because the status of a schematic variable tells us how much computation is safe. Trusted things — inputs and outputs — may reduce arbitrarily; untrusted things — subjects — may take at most one step. A thing which may have reduced arbitrarily never appears in a subject position where computation is restricted, so the induction hypothesis for the conclusion will always apply: if the conclusion subject takes at most one step, then each of the premise subjects — its components — takes at most one step.

We may now *state* a sensible formulation of *subject reduction*. Indeed, we may *compute* it from the judgement forms. Suppose an 'old' judgement holds; compute its context and inputs arbitrarily and its subjects by at most one step; there exist new outputs computable somehow from the old outputs such that the new judgment holds.

Definition 35 (*subject reduction*)
The property of *subject reduction* is said to hold if the following are admissible.

$$\frac{\Gamma \vdash \mathrm{TYPE}\ T \quad \Gamma \rightsquigarrow^* \Gamma' \quad T \rightsquigarrow^? T'}{\Gamma' \vdash \mathrm{TYPE}\ T'}$$

$$\frac{\Gamma \vdash T \ni t \quad \Gamma \rightsquigarrow^* \Gamma' \quad T \rightsquigarrow^* T' \quad t \rightsquigarrow^? t'}{\Gamma' \vdash T' \ni t'} \qquad \frac{\Gamma \vdash e \in S \quad \Gamma \rightsquigarrow^* \Gamma' \quad e \rightsquigarrow^? e'}{\exists S'.\, \Gamma' \vdash e' \in S' \wedge S \rightsquigarrow^* S'}$$

In order to show this property, we shall need to say how to formulate the rules and check that a suitable condition holds of them.

Let us now design the language of patterns, then consider how to manage the business of binding schematic variables for use in expressions.

6.2 There are some questions it is better not to ask

It is the job of a formal pattern to ask questions about an actual term. 'What is a sensible question?' is a sensible meta-question. For example, 'Are you the free variable x?' is a *dangerous* question, because the answer is 'Yes.' for the free variable x, but perhaps not for some substitution instance of x. As Dogma 25 indicates, our rules should not ask such questions if we want typing derivations to be stable under substitution.

We shall have some language of patterns (p, q) which come equipped with a notion of p-environment (π, χ) mapping schematic variables in p to actual terms, such that any actual term 'matching' the pattern p will given by the action, πp, for some p-environment π. The grammar of possible ps will be some restriction of the grammar of actual terms. We had better ensure that environments can be acted on by thinnings, $\pi\theta$, and by substitutions, $\pi\sigma$. Moreover, we must guarantee that

$$(\pi p)\theta = (\pi\theta)p \qquad (\pi p)\sigma = (\pi\sigma)p$$

The latter should remind us to ask whether there are any syntactic constructs, other than free variables, which are *not* preserved by substitution, and in our setup, we should remember that

$$\underline{x}/(t:T) = t$$

from which we deduce that 'Are you a thunk?' is also a dangerous question for a pattern to ask.

Remark 36 (*the* υ *alternative*)
In previous work, I have proposed to permit terms of form $\underline{(t:T)}$ and hence to allow structural substitution for thunks, at the cost of adopting the additional reduction rule,

$$\underline{(t:T)} \rightsquigarrow_{\upsilon} t$$

where υ is the grunt of satisfaction at the complete elimination of a cut. However, such a rearrangement does not affect the sensible notion of pattern, because of the following consideration.

Dogma 37 (*stability of pattern matching under reduction*)
Consider pattern p. If $t = \pi p$ for some p-environment, π, and $t \rightsquigarrow t'$, then there must exist some π' such that $t' = \pi' p$ and $\pi \rightsquigarrow \pi'$, where the latter is the extension of reduction to environments which allows reduction in exactly one term therein.

Proof
Given Definition 28, we shall shortly ensure this property via Definition 38. □

Why should we adopt such a dogma? Given that we may precompute types before checking them by pattern matching and postcompute synthesized types before analysing them, again by pattern matching, it would be awkward, to say the least, if positive matches were transient. It is one thing to say that some computation must be done before a match is achieved, but quite another to say that some computation must be avoided to ensure that a match is sustained — 'blink and you'll miss it'. There are two properties which are reliably stabilised by computation:

- being an *essential* construction (or 'canonical form')
- *independence* of a variable

Independence of free variables is not stable under substitution, but independence of *bound* variables is unproblematic, as they are unaffected by substitution. Accordingly, we have found a class of syntactic questions which a pattern may reasonably ask. Let us define their grammar.

6.3 Patterns

Definition 38 (γ*-pattern*)
The grammar of patterns, $\mathsf{Pat}\,\gamma$, for a given scope is as follows:

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

where $\bot$ is the pattern which *never* matches, and we identify

$$\bot = (\bot . q) = (p . \bot) = \backslash x \bot$$

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

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

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 $\cdot\theta : \mathsf{Pat}\,\gamma \to \mathsf{Pat}\,\gamma'$, going under binders in the same way as for terms, and acting on placeholders by composition. The effect of this is to grow the notional scope of a pattern while forbidding any of the inserted variables to occur in its 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

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

If we wish a different thinning, we may attach angle brackets to the name, listing the permitted variables. We could also have written:

$$(\Pi\, S\langle\rangle \setminus x\, T\langle x\rangle)$$

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

The $\bot$ pattern may seem a trifle peculiar: it would certainly be rather pointless to use it in a rule, as that would ensure its inapplicability. My motivation for introducing it is primarily algebraic. Patterns have a sensible notion of *refinement*: informally, $p \sqsubseteq p'$ if everything matching p also matches p'. The top of this ordering is the identity thinning $\mathbf{1}$, and $\bot$ is its bottom.

Definition 39 (*pattern refinement*)
Pattern refinement is given inductively as follows.

$$\frac{}{a \sqsubseteq a} \qquad \frac{p \sqsubseteq p' \quad q \sqsubseteq q'}{(p\,.\,q) \quad (p'\,.\,q')} \qquad \frac{q \sqsubseteq q'}{\setminus x\, q \sqsubseteq \setminus x\, q'} \qquad \frac{}{p\theta \sqsubseteq \theta} \qquad \frac{}{\bot \sqsubseteq p}$$

In due course, we shall establish that refinement makes patterns a category with *pull-backs*, giving us the notion of most general unifier that we need.

Now, to make patterns talk about terms, we need the environments, π, that explain what 'matching' means.

Definition 40 (*matching p-environment*)
A pattern $p : \mathsf{Pat}\,\gamma$ induces a set of p-environments, whose name we may abbreviate to p, defined inductively as follows.

$$\frac{}{a : a} \qquad \frac{\pi : p \quad \chi : q}{(\pi\,.\,\chi) : (p\,.\,q)} \qquad \frac{\chi : q}{\setminus x\, \chi : \setminus x\, q} \qquad \frac{t : \mathsf{Term}\,\mathsf{lib}\,\mathsf{cons}\,\delta \quad \theta : \delta \sqsubseteq \gamma}{t : \theta}$$

Such a p-environment, π, acts on p to give

$$\pi p : \mathsf{Term}\,\mathsf{lib}\,\mathsf{cons}\,\gamma$$

where, in particular, the action $t\theta$ is the term $t\theta$. A term *matches* p if it can be expressed as πp for some π.

Speaking formally, what then, is a *schematic variable*? As patterns constitute the binding sites of schematic variables, we may consider latter to be given by *paths* to placeholders.

Definition 41 (*schematic p-variable, projection from environment*)

If $p : \mathsf{Pat}\,\gamma$, let $\delta \leftarrow p$ be the set of schematic p-variables, ξ, with scope δ, defined inductively as follows.

$$\frac{\theta : \delta \sqsubseteq \gamma}{\bullet : \delta \leftarrow \theta} \qquad \frac{\xi : \delta \leftarrow p}{(\xi\,.) : \delta \leftarrow (p\,.\,q)} \qquad \frac{\xi : \delta \leftarrow q}{(.\,\xi) : \delta \leftarrow (p\,.\,q)} \qquad \frac{\xi : \delta \leftarrow q}{\backslash\,\xi : \delta \leftarrow \backslash x\,q}$$

Moreover, given such a ξ and some $\pi : p$, we obtain $\xi\,!\,\pi : \mathsf{Term}\,\mathsf{lib}\,\mathsf{cons}\,\delta$ by following the path.

Informally, of course, we shall continue to use *names* for schematic variables. As you can see where each name stands in the pattern which binds it, you can tell what its formal path must be.

Computation patterns So far, we have considered patterns for *constructions*. However, we have seen in our examples, particularly the elimination rule for `cdr`, that we have need of schematic variables which stand for *computations*. We have learned also that no finer analysis of computations is stable. Accordingly, schematic variables are the only sensible patterns for computations, naming the whole of them. We may thus keep track of them by reusing the notion of scope from the underlying term syntax. Let us, however, keep a clear separation between the scope of schematic variables in an expression and the scope of its term variables.

6.4 Expressions

Let us now define the notion of *expression* which appears in formal judgements, with a precise notion of *schematic scope*.

Definition 42 (*schematic scope*)

A schematic scope is a pair $(\delta|p)$, where δ is an ordinary scope, giving the schematic variables which stand for computations, and $p : \mathsf{Pat}\,\varepsilon$, determining the schematic variables which stand for constructions.

Definition 43 (*expression*)

The syntax of expressions $\mathsf{Expr}\,(\delta|p)\,l\,d\,\gamma$, with schematic scope $(\delta|p)$ and scope γ extends that of terms, as follows:

$d \setminus l$	essential	liberal
construction	k_γ, K_γ	$s_\gamma, t_\gamma, S_\gamma, T_\gamma$
	$::= \quad a$	$::= \quad k_\gamma$
	$\mid \quad (s_\gamma\,.\,t_\gamma)$	$\mid \quad n_\gamma$
	$\mid \quad \backslash x\, t_{\gamma,x}$	$\mid \quad \overline{\xi}/\sigma \quad$ where $\xi : \gamma' \leftarrow p, \sigma : (\mathsf{Expr}\,(\delta|p)\,\mathsf{lib\,comp}\,\gamma)^{\gamma'}$
computation	n_γ, N_γ	$e_\gamma, f_\gamma, E_\gamma, F_\gamma$
	$::= \quad x_\gamma$	$::= \quad n_\gamma$
	$\mid \quad e_\gamma\, s_\gamma$	$\mid \quad (t_\gamma : T_\gamma)$
	$\mid \quad x_\delta$	

In particular, $\mathsf{Term}\,l\,d\,\gamma$ is now seen to be $\mathsf{Expr}\,(\varepsilon, ())\,l\,d\,\gamma$.

That is, the essential computations are extended with schematic computation variables from δ, and the liberal constructions are extended with *instantiations* of schematic p-variables by a substitution of the term variables upon which they may depend, hence our commonplace $T/(s:S)$. By informal convention, we may abbreviate ξ/ι to ξ, which means that if some T is bound in a pattern $\backslash x\,T$, we may write T as an expression in any scope containing x, as we have already seen in the rule for checking a function:

$$\frac{x : S \vdash T \ni t}{(\Pi\, S\ \backslash x\, T) \ni \backslash x\, t}$$

The binding site of T is in the rule's conclusion: the checked type T in the premise is really abbreviating the expression T/x, with the x in the premise's context extension effectively capturing the bound x in T.

6.5 Premises

Now that we know what an expression is, we can say what a *premise* is. The key point is that premises are defined with respect to a schematic scope of *trusted* schematic variables and a pattern of *untrusted* schematic variables. A type construction or type checking premise will select one of the untrusted schematic variables as its subject and establish trust in it. Each such premise must grow the trusted schematic scope and remove the subject from the untrusted pattern, until everything is trusted.

Definition 44 (*pattern variable removal, placelessness*)
If $q : \mathsf{Pat}\,\gamma$ and $\xi : \delta \leftarrow q$, then $q - \xi$ is given by replacing the placeholder in q pointed to by ξ with the atom `()`. Likewise, if $\chi : q$, we may take $\chi - \xi$ to be the environment in $q - \xi$ given by replacing the term in χ pointed to by ξ with the atom `()`. A pattern with no placeholders is said to be *placeless*.

Definition 45 (*formal premise*)
If $(\delta|p)$ is the schematic scope of trusted schematic variables, $q : \mathsf{Pat}\,\varepsilon$ is the pattern of untrusted schematic variables, and γ is the scope of term variables, then $\mathsf{Prem}\,(\delta|p)\,q\,\gamma\,p'\,q'$

is the set of valid premises establishing *fresh* trust in $p' : \mathsf{Pat}\,\gamma$ and leaving $q' : \mathsf{Pat}\,\varepsilon$ untrusted, defined inductively as follows:

$$\frac{\xi : \delta \leftarrow q \quad \theta : \delta \sqsubseteq \gamma}{\textsc{type}\ \xi\theta : \mathsf{Prem}\,(\delta|p)\,q\,\gamma\,\theta\,(q - \xi)}$$

$$\frac{T : \mathsf{Expr}\,(\delta|p)\,\mathsf{lib}\,\mathsf{cons}\,\gamma \quad \xi : \delta \leftarrow q \quad \theta : \delta \sqsubseteq \gamma}{T \ni \xi\theta : \mathsf{Prem}\,(\delta|p)\,q\,\gamma\,\theta\,(q - \xi)}$$

$$\frac{S, T : \mathsf{Expr}\,(\delta|p)\,\mathsf{lib}\,\mathsf{cons}\,\gamma}{S = T : \mathsf{Prem}\,(\delta|p)\,q\,\gamma\,()\,q} \qquad \frac{T : \mathsf{Expr}\,(\delta|p)\,\mathsf{lib}\,\mathsf{cons}\,\gamma}{\textsc{univ}\ T : \mathsf{Prem}\,(\delta|p)\,q\,\gamma\,()\,q}$$

$$\frac{S : \mathsf{Expr}\,(\delta|p)\,\mathsf{lib}\,\mathsf{cons}\,\gamma \quad J : \mathsf{Prem}\,(\delta|p)\,q\,(\gamma, x)\,p'\,q'}{x{:}S \vdash J : \mathsf{Prem}\,(\delta|p)\,q\,\gamma\,(\backslash x\,p')\,q'}$$

Before we walk through this definition, let us see how it plugs into the definition of a formal premise sequence, managing scope correctly.

Definition 46 (*formal premise sequence*)
If $(\delta|p)$ is the schematic scope of trusted schematic variables, $q : \mathsf{Pat}\,\varepsilon$ is the pattern of untrusted schematic variables, then $\mathsf{Prems}\,(\delta|p)\,q\,p'$ is the set of valid premise sequences establishing *overall* trust in $p' : \mathsf{Pat}\,\varepsilon$, defined inductively.

$$\frac{q\ \text{placeless}}{\varepsilon : \mathsf{Prems}\,(\delta|p)\,q\,p} \qquad \frac{J : \mathsf{Prem}\,(\delta|p_0)\,q_0\,\varepsilon\,p'\,q_1 \quad \vec{J} : \mathsf{Prems}\,(\delta|(p_0\,.\,p'))\,q_1\,p_2}{J\,\vec{J} : \mathsf{Prems}\,(\delta|p_0)\,q_0\,p_2}$$

There is quite a lot to take in, here, so we shall proceed slowly and carefully.

Firstly, note how the untrusted qs thread left-to-right. The empty sequence demands that q be placeless, ensuring that we stop only once trust has been established in everything. The only place where q changes is in a *validation* premise: a subject $\xi : \delta \leftarrow q$ is selected, and the untrusted pattern $q - \xi$ is sent onwards and rightwards.

Secondly, notice how the trusted ps thread left-to-right. The definition of *premise* gives us a p' which has freshly become trustworthy because of that premise. Correspondingly, in the step case of sequences, the trusted pattern for the tail is $(p_0\,.\,p')$, where p_0 is what was already trusted and p' is what the head premise has validated. Observe that validation premises give us a new trusted schematic variable — the validated subject — while a test gives us nothing new. Furthermore, whenever a premise contains an expression, it depends only on trusted schematic variables.

Thirdly, let us consider what happens with the scope γ of term variables. A premise sequence consists of ε-premises: we cannot talk about free variables, reifying Dogma 25. A premise can, however, locally extend the scope by means of context extension, with a suitable expression giving the type of the bound variable. If we eventually arrive at a validation, its subject $\xi\theta$ uses θ to select which of those bound variables are the variables ξ depends on. That is, we insist ξ is validated in full generality, not just for a particular substitution instance. Informally, of course, we never need to write this thinning, because we choose the names in the context extension to capture the names free at the subject's binding site.

Fourthly, you are entitled to some concern that there are no type *synthesis* premises. That is because elimination rules need premises only to validate components of the *eliminator*,

which are all constructions. The validation of the *target*, synthesizing its type, can be treated uniformly, as we shall soon see.

6.6 What are the rules?

Let us now work through the judgement forms, characterizing the rules which we may write for each.

Definition 47 (*formal type construction*)
A *type construction* rule takes the form

$$\frac{\vec{J}}{\text{TYPE}\ q}$$

where $q : \mathsf{Pat}\,\varepsilon$ and $\vec{J} : \mathsf{Prems}\,(\varepsilon|\,()\,)\,q\,p'$ for some p'.

That is, we initially trust *nothing*, but we learn to trust the things in q as we work our way through $\vec{J}$.

Definition 48 (*formal type checking*)
A *type checking* rule takes the form

$$\frac{\vec{J}}{p \ni q}$$

where $p, q : \mathsf{Pat}\,\varepsilon$ and $\vec{J} : \mathsf{Prems}\,(\varepsilon|p)\,q\,p'$ for some p'.

That is, we initially trust the components of the *type*, and we learn to trust the components of the term.

Definition 49 (*formal elimination rule*)
The *type synthesis* rule for an elimination takes the form

$$\frac{e \in p \quad \vec{J}}{e\,q \in S}$$

where $p, q : \mathsf{Pat}\,\varepsilon$, $\vec{J} : \mathsf{Prems}\,(e|p)\,q\,p'$ for some p', and $S : \mathsf{Expr}\,(e|p')\,\mathsf{lib}\,\mathsf{cons}\,\varepsilon$.

That is, we first synthesize the type of the target, e and match it with a pattern, p, whose components we trust because of the type synthesis postcondition. Meanwhile, we have matched the eliminator with a pattern, q. We may now check the eliminator with a premise sequence trusting p and also the target, e — our one schematic computation variable. By the time we have validated the eliminator, we now trust p', so the synthesized type may refer to e and to the components of p'.

Definition 50 (*formal universe checking*)
The rules for checking whether a known type is a *universe* take the form

$$\frac{\vec{J}}{\text{UNIV}\ p}$$

where $p : \mathsf{Pat}\,\varepsilon$ and $\vec{J} : \mathsf{Prems}\,(\varepsilon|p)\,()\,p'$ for some p'.

Here, we already trust the components of the type, so there is nothing untrusted.

Example 51 (*Martin-Löf's 1971 theory*)
By inspection, the configured rules for our reconstruction of Martin-Löf's 1971 theory (see Section 5.4) are admitted by the above definitions.

6.7 How to apply a rule

We have said how to construct the formal rules from *formal* judgements. Let us now say how to deploy those rules to derive *actual* judgements. In particular, even though the rules never talk about free variables, the actual terms they talk about may very well contain free variables.

How are we to allow patterns $p : \mathsf{Pat}\,\varepsilon$ to refer to terms over some γ? How are we to instantiate expressions with the resulting environments? The first step is to define the *opening* of a pattern with respect to a scope.

Definition 52 (*pattern opening*)
If $p : \mathsf{Pat}\,\delta$, let $\gamma \otimes p : \mathsf{Pat}\,(\gamma, \delta)$ given by replacing each θ in p with $\mathbf{1}_\gamma, \theta$.

Correspondingly, if $\pi : \gamma \otimes p$, then the term in π which instantiates some $\xi : \delta' \leftarrow p$ will have scope γ, δ'. That is to say, it may use free variables, along with those of the locally bound variables selected by its placeholder. A term $t : \mathsf{Term}\,\mathsf{lib}\,\mathsf{cons}\,(\gamma, \delta)$ matches a pattern $p : \mathsf{Pat}\,\delta$ if it can be expressed as $\pi(\gamma \otimes p)$ for some environment π.

A schematic scope $(\delta|p)$ can thus be instantiated in scope γ by a pair $(\rho|\pi)$ where $\rho : \delta \Rightarrow \gamma$ and $\pi : \gamma \otimes p$.

Definition 53 (*instantiation*)
Let $(\delta|p)$ be a schematic scope and γ some target scope of free variables. Let $\rho : \delta \Rightarrow \gamma$ and $\pi : \gamma \otimes p$. We may define an action

$$\cdot(\rho|\pi) : \mathsf{Expr}\,(\delta|p)\,l\,d\,\gamma' \to \mathsf{Term}\,\mathsf{lib}\,d\,\gamma, \gamma'$$

The action proceeds structurally, extending γ' to go under a binder without modifying ρ or π. We need only explain what happens at variables and schematic variables:

- A term variable x in γ' becomes $x(\gamma \triangleright \gamma')$.
- A schematic computation variable x in δ becomes $x\rho(\gamma \triangleleft \gamma'$.
- For $\xi\,/\,\sigma$, with $\xi : \delta' \leftarrow p$ and $\sigma : (\mathsf{Expr}\,(\delta|p)\,l\,d\,\gamma')^{\delta'}$, we first obtain

  $$\xi\,!\pi : \mathsf{Term}\,\mathsf{lib}\,\mathsf{cons}\,\gamma, \delta' \qquad \sigma(\rho|\pi) : \delta' \Rightarrow \gamma, \gamma'$$

  We then substitute

  $$(\xi\,!\pi)(\iota(\gamma \triangleleft \gamma'), \sigma(\rho|\pi))$$

  which is a precise way of saying that we project ξ's term from the π and substitute its bound variables with the instantiation of σ, letting its free variables be.

To deploy this machinery, we need to follow the structure of Definitions 45 and 46, repeating the same juggling act for trusted and untrusted environments, π and χ, respectively, as we did between the trusted and untrusted patterns, p and q. Fortunately, there is only one way to do it.

Definition 54 (*actual premise*)
If $J : \mathsf{Prem}\,(\delta|p)\,q\,\gamma\,p'\,q'$, $\rho : \delta \Rightarrow \gamma'$, $\pi : \gamma' \otimes p$, and $\chi : \gamma' \otimes q$, then define

$$\mathsf{prem}\,J\,(\rho|\pi)\,\chi = J'|\pi'|\chi' \text{ where } J' \text{ is a } \gamma', \gamma\text{-judgment, } \pi' : \gamma' \otimes p' \text{ and } \chi' : \gamma' \otimes q'$$

whose computation proceeds as follows:

$$\begin{array}{lllll}
\mathsf{prem}\,(\mathrm{TYPE}\,\xi\,\theta)\;(\rho|\pi)\,\chi & = & \mathrm{TYPE}\,(\xi\,!\,\chi)(\mathbf{1}_{\gamma'},\theta) & |\,\xi\,!\,\chi & |\,\chi - \xi \\
\mathsf{prem}\,(T \ni \xi\,\theta) & (\rho|\pi)\,\chi & = & T(\rho|\pi) \ni (\xi\,!\,\chi)(\mathbf{1}_{\gamma'},\theta) & |\,\xi\,!\,\chi\;|\,\chi - \xi \\
\mathsf{prem}\,(S = T) & (\rho|\pi)\,\chi & = & S(\rho|\pi) = T(\rho|\pi) & |\,() \quad |\,\chi \\
\mathsf{prem}\,(\mathrm{UNIV}\,T) & (\rho|\pi)\,\chi & = & \mathrm{UNIV}\,T(\rho|\pi) & |\,() \quad |\,\chi \\
\mathsf{prem}\,(x{:}S \vdash J) & (\rho|\pi)\,\chi & = & x{:}S(\rho|\pi) \vdash J' & |\,\backslash x\,\pi'\,|\,\chi' \\
\multicolumn{5}{l}{\quad \text{where } J'|\pi'|\chi' = \mathsf{prem}\,J\,(\rho|\pi)\,\chi}
\end{array}$$

That is, the schematic variables are everywhere instantiated using the trusted environment, and the validation premises move the validated term from the untrusted environment to the freshly trusted environment. We may now iterate across a formal premise sequence.

Definition 55 (*actual premise sequence*)
If $\vec{J} : \mathsf{Prems}\,(\delta|p)\,q\,p'$, $\rho : \delta \Rightarrow \gamma'$, $\pi : \gamma' \otimes p$, and $\chi : \gamma' \otimes q$, then define

$$\mathsf{prems}\,\vec{J}\,(\rho|\pi)\,\chi = \vec{J}'|\pi'|\chi' \text{ where } \vec{J}' \text{ is a list of } \gamma', \gamma\text{-judgments, and } \pi' : \gamma' \otimes p'$$

whose computation proceeds as follows.

$$\begin{array}{lll}
\mathsf{prems}\,\varepsilon \quad (\rho|\pi)\,\chi & = & \varepsilon \quad |\,\pi \\
\mathsf{prems}\,J\,\vec{J}\,(\rho|\pi_0)\,\chi_0 & = & J'\;\vec{J}'\,|\,\pi_2 \\
\quad \text{where} \quad J'|\pi'|\chi_1 = \mathsf{prem}\,J\,(\rho|\pi_0)\,\chi_0 & & \vec{J}'|\pi_2 = \mathsf{prems}\,\vec{J}\,(\rho|(\pi_0\,.\,\pi'))\,\chi_1
\end{array}$$

We can now say which actual derivations our formal rules are willing to yield.

Definition 56 (*actual rules*)

formal	**actual**	
$\dfrac{\vec{J}}{\mathrm{TYPE}\,q}$	$\dfrac{\vec{J}'}{\mathrm{TYPE}\,\chi q}$	$\vec{J}'\|\pi' = \mathsf{prems}\,\vec{J}\,(\varepsilon|())\,\chi$
$\dfrac{\vec{J}}{p \ni q}$	$\dfrac{\vec{J}'}{\pi p \ni \chi q}$	$\vec{J}'\|\pi' = \mathsf{prems}\,\vec{J}\,(\varepsilon|\pi)\,\chi$
$\dfrac{e \in p \quad \vec{J}}{e\,q \in S}$	$\dfrac{e \in \pi p \quad \vec{J}'}{e\,(\chi q) \in S(e|\pi')}$	$\vec{J}'\|\pi' = \mathsf{prems}\,\vec{J}\,(e|\pi)\,\chi$
$\dfrac{\vec{J}}{\mathrm{UNIV}\,p}$	$\dfrac{\vec{J}'}{\mathrm{UNIV}\,\pi p}$	$\vec{J}'\|\pi' = \mathsf{prems}\,\vec{J}\,(\varepsilon|\pi)\,()$

Only one component is now missing. Let us fill it in, forthwith!

Definition 57 (*formal and actual β-reduction rules*)
A formal β-rule is given by a quintuple comprising three patterns and two expressions

$$(p_0 : p_1)\;p_2 \rightsquigarrow_\beta (t : T)$$

where $p_i : \mathsf{Pat}\,\varepsilon$ and $t, T : \mathsf{Expr}\,(\varepsilon|()\,.\,(p_0\,.\,p_1))\,p_2\,\mathsf{lib}\,\mathsf{cons}\,\varepsilon$. The actual β-rule thus represented is

$$(\pi_0 p_0 : \pi_1 p_1)\;(\pi_2 p_2) \rightsquigarrow_\beta (t(\varepsilon|((\pi_0\,.\,\pi_1)\,.\,\pi_2)) : T(\varepsilon|((\pi_0\,.\,\pi_1)\,.\,\pi_2)))$$

6.8 What is a bidirectional type theory?

We are, at last, in a position to specify what is meant by 'specifying a bidirectional type theory', in the sense of this paper.

Definition 58 (*bidirectional type theory*)
A bidirectional type theory is presented by giving the sets of formal rules for type construction, type checking, elimination and universe checking, as specified in Definitions 47 through 50, together with a set of formal β-reduction rules as specified in Definition 57. The rules of the type theory thus presented are as follows: EXTEND and LOOKUP (Definition 22); VAR (Definition 24); THUNK, UNIVERSE and RADICAL (Definition 26); PRE and POST (Definition 27); the actual translations of the formal rules given in Definition 56. Its reduction rules are given by the contextual closure rules in Definition 28 together with the actual translation of the formal β-rules given by Definition 57.

The point of making such a definition is to establish metatheoretical properties of the entire class of bidirectional type theories. In order to do so, we shall need to go back and identify the necessary structural properties of the equipment we have built thus far.

7 Stability of derivations under thinning and substitution

In as much as the expressions we write in our formal rules make use of substitution, whether in typing rules or in β-rules (and our example system does both), the metatheoretic health of our theories will rest on the stability of derivations under substitutions which are suitably type-preserving. Fortunately, we have done the hard work to ensure that this is so. But what is so? Let us put the claim on a formal footing and develop the structure required to prove it.

We can say what it means for a thinning to be *contextual*.

Definition 59 (*contextual thinning*)
If $\theta : \gamma \sqsubseteq \gamma'$, Γ is a γ-context and Γ' is a γ'-context, then we say θ is a *contextual* thinning from Γ to Γ' and write

$$\Gamma \sqsubseteq_\theta \Gamma' \quad \text{whenever} \quad \Gamma\theta = \theta!\Gamma'$$

Note that the above is a relatively weak definition: it certainly does not guarantee that Γ' will be *valid*, even if Γ is. It does just enough work to ensure that Γ' and Γ agree on the types of the free variables related by θ.

We can further say what it means for a substitution to be contextual.

Definition 60 (*contextual substitution*)
If $\sigma : \gamma \sqsubseteq \gamma'$, Γ is a γ-context and Γ' is a γ'-context, then we say σ is a *contextual* substitution from Γ to Γ' and write

$$\Gamma \Rightarrow_\sigma \Gamma' \quad \text{whenever} \quad \Gamma' \vdash x!\sigma \in (x!\Gamma)\sigma \text{ is derivable for all } x \text{ in } \gamma$$

That is, σ must map variables to computations such that the *advertised* type of each variable can be *synthesized* for the corresponding computation. We expect to replace appeals to the VAR rule by the given derivations, and we shall need these to be stable under contextual thinnings if we are to pass under binders.

On the one hand, we shall need to ensure that the fixed part of the bidirectional setup satisifies these properties. On the other hand, we shall need to check that the instantiation of formal rules to actual rules absorbs thinnings and substitutions, so that stability amounts to choosing a thinned or substituted instance of the same formal rules invoked in the derivation that we seek to preserve.

In short, we must establish that our various actions on syntax are appropriately compositional.

7.1 The monoidal category of thinnings acts functorially on syntax

Thinnings form a well known category: the *semi-simplicial* category, often notated Δ_+.

Lemma 61 (*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 62 (*monoidal structure of thinnings*)
Concatenation of bit vectors, θ, ϕ, induces monoidal structure on Δ_+ with respect to the monoid of concatenation on scopes. That is, if $\theta : \gamma \sqsubseteq \delta$ and $\phi : \gamma' \sqsubseteq \delta'$, then $\theta \otimes \phi : \gamma, \gamma' \sqsubseteq \delta, \delta'$ is given by θ, ϕ, satisfying

$$\mathbf{1} \otimes \mathbf{1} = \mathbf{1} \qquad (\theta\theta') \otimes (\phi\phi') = (\theta \otimes \phi)(\theta' \otimes \phi')$$

such that

$$\mathbf{1} \otimes \theta = \theta = \theta \otimes \mathbf{1} \qquad (\theta_0 \otimes \theta_1) \otimes \theta_2 = \theta_0 \otimes (\theta_1 \otimes \theta_2)$$

Proof
These equations are readily shown by functional induction on concatenation of thinnings. □

Lemma 63 (*functoriality of the thinning action*)
The thinning action extends $\mathsf{Term}\, l\, d\, \cdot$ 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 $\cdot \otimes 1$ to pass under an abstraction. □

Remark 64 (*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. Consider 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.

7.2 Thinnings act contravariantly on vectors

Lemma 65 (*natural selection*)
The selection operator $\cdot!\cdot$ extends $X^{\cdot}$ to a functor from Δ_+^{op} to **Set**. Moreover $\theta!\cdot$ is a natural transformation from $\cdot^{\gamma}$ to $\cdot^{\delta}$.

Proof
We require

$$\mathbf{1}!\vec{x} = \vec{x} \qquad (\theta\phi)!\vec{x} = \theta!(\phi!\vec{x}) \qquad \theta!(\vec{x}f) = (\theta!\vec{x})f$$

where postfix $\cdot f$ is the pointwise mapping of $f : X \to Y$ across a vector. All are readily established by functional induction. □

Selection by injections allows us to take prefixes and suffixes of vectors. In particular, we can chop a vector in two, then stick it back together.

Lemma 66 (*chopstick*)
If $\vec{x} : X^{\gamma,\delta}$, then

$$(\gamma \triangleleft \delta)!\vec{x}, (\gamma \triangleright \delta)!\vec{x} = \vec{x}$$

Proof
This follows by functional induction on concatenation of scopes. □

7.3 The monoidal category of substitutions acts functorially on syntax

Unlike with thinnings, concatenation is not quite enough to induce monoidal structure, and has the wrong type to be the tensor. We shall need to fix up the target scopes with some thinnings.

Definition 67 (*tensor of substitutions, weakening*)
If $\sigma_i : \gamma_i \Rightarrow \delta_i$ for $i = 0, 1$, we may take

$$\sigma_0 \otimes \sigma_1 : \gamma_0, \gamma_1 \Rightarrow \delta_0, \delta_1 \quad \sigma_0 \otimes \sigma_1 = \sigma_0(\delta_0 \triangleleft \delta_1), \sigma_1(\delta_0 \triangleright \delta_1)$$

In particular, if $\sigma : \gamma \Rightarrow \delta$, then its weakening, $\sigma \otimes 1 : \gamma, x \Rightarrow \delta, x$, is given by

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

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'

Lemma 68 (*action of identity*)
If $\theta : \gamma \sqsubseteq \delta$, then

$$\theta!\iota_\delta = \iota_\gamma\theta : \gamma \Rightarrow \delta$$

As a consequence, $x\iota = x$.

Proof
Induction on scopes establishes the former; induction on x, the latter. □

Lemma 69 (*action of compositions*)

All four compositions of thinnings and substitions act on terms as the sequence of the component actions.

$$t(\theta\phi) = (t\theta)\phi \quad t(\theta!\sigma) = (t\theta)\sigma \quad t(\rho\phi) = (t\rho)\phi \quad t(\rho\sigma) = (t\rho)\sigma$$

Accordingly, it is unambiguous to abbreviate $\theta!\sigma$ by $\theta\sigma$ for selection from a vector which happens to be a substitution.

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

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

for both thinnings and substitutions. □

We may now readily establish that substitutions form a category.

Lemma 70 (*category of substitutions*)
Substitutions $\sigma : \gamma \Rightarrow \delta$ are the arrows of a category with identity ι and pointwise composition. Further, action on terms extends $\mathsf{Term}\,\mathsf{lib}\,d\,\cdot$ to functor from substitutions to **Set**.

Proof
Pointwise lifting of Lemmas 68 and 69 establishes the category, which allows us to state the functoriality those Lemmas already embody. □

It is, moreover, a *monoidal* category.

Lemma 71 (*monoidal structure of substitutions*)
Substitutions are a monoidal category with unit the trivial substitution between empty scopes, and tensor $\otimes$.

Proof
The monoidal properties of $\otimes$ follow by plumbing with thinnings. □

Lemma 72 (*monoidal functor from thinnings to substitutions*)
Composition $\cdot\iota = \iota\cdot$ is the action on arrows of an identity-on-objects monoidal functor from thinnings to substitutions.

Proof
We already have that $\mathbf{1}\iota = \iota\mathbf{1} = \iota$. We may readily see that

$$(\theta\iota)(\phi\iota) = \iota(\theta(\phi\iota)) = (\theta(\phi\iota)) = (\theta\phi)\iota$$

It takes a little more work to establish that when $\theta_i : \gamma_i \sqsubseteq \delta_i$

$$\begin{array}{rcll}(\theta_0\iota) \otimes (\theta_1\iota) & = & (\iota\theta_0) \otimes (\iota\theta_1) & \\ & = & \iota\theta_0(\delta_0 \lhd \delta_1), \iota\theta_1(\delta_0 \rhd \delta_1) & \\ & = & \iota(\delta_0 \lhd \delta_1)(\theta_0 \otimes \theta_1), \iota(\delta_0 \rhd \delta_1)(\theta_0 \otimes \theta_1) & \\ & = & ((\delta_0 \lhd \delta_1)\iota, (\delta_0 \rhd \delta_1)\iota)(\theta_0 \otimes \theta_1) & \\ & = & \iota(\theta_0 \otimes \theta_1) & = \quad (\theta_0 \otimes \theta_1)\iota\end{array}$$

□

7.4 Action of thinnings and substitutions on environments

7.5 Stability of typing derivations under contextual thinnings and substitutions

8 Pullbacks of patterns put to purpose

Lemma 73 (Δ_+ *has pullbacks*)
If $\theta_i : \gamma_i \sqsubseteq \gamma$ for $i = 0,1$, then there exists a scope δ, a thinning $\theta : \delta \sqsubseteq \gamma$, and thinnings $\phi_i : \delta \sqsubseteq \gamma_i$ such that $\phi_i\theta_i = \theta$ for $i = 0,1$ with the universal property that for any $\phi : \delta' \sqsubseteq \gamma$ and $\phi'_i : \delta' \sqsubseteq \gamma_i$ such that $\phi = \phi'_0\theta_0$, we have some $\theta' : \delta' \sqsubseteq \delta$ such that $\phi = \theta'\theta$ and $\phi'_i = \theta'\phi_i$ for $i = 0,1$. We define $\theta_0 \sqcap \theta_1 = \phi_0|\theta|\phi_1$, leaving δ implicit.

Proof
We shall see that θ is the pointwise conjunction of the θ_i, which needs must embed in both γ_i. More formally, we compute pullbacks thus:

$$\begin{array}{rcll}
\varepsilon \sqcap \varepsilon & = & \varepsilon|\ \varepsilon\ |\varepsilon & \\
(\theta_0,0) \sqcap (\theta_1,0) & = & \phi_0|\theta,0|\phi_1 & \text{where } \phi_0|\theta|\phi_1 = \theta_0 \sqcap \theta_1 \\
(\theta_0,0) \sqcap (\theta_1,1) & = & \phi_0|\theta,0|\phi_1,0 & \text{where } \phi_0|\theta|\phi_1 = \theta_0 \sqcap \theta_1 \\
(\theta_0,1) \sqcap (\theta_1,0) & = & \phi_0,0|\theta,0|\phi_1 & \text{where } \phi_0|\theta|\phi_1 = \theta_0 \sqcap \theta_1 \\
(\theta_0,1) \sqcap (\theta_1,1) & = & \phi_0,1|\theta,1|\phi_1,1 & \text{where } \phi_0|\theta|\phi_1 = \theta_0 \sqcap \theta_1
\end{array}$$

The universal property holds by straightforward induction on the call graph of $\cdot \sqcap \cdot$. □

Remark 74 (*pullback by selection*)
If we view $\theta_i : \gamma_i \sqsubseteq \gamma$ for $i = 0,1$ as bit vectors of length γ, it becomes reasonable to compute $\theta_0!\theta_1 : \delta \sqsubseteq \gamma_0$ and $\theta_1!\theta_0 : \delta \sqsubseteq \gamma_1$. These are, respectively, ϕ_0 and ϕ_1 whenever $\theta_0 \sqcap \theta_1 = \phi_0|\theta|\phi_1$.

Lemma 75 (Δ_+/γ *has coproducts*)
If $\theta_i : \gamma_i \sqsubseteq \gamma$ for $i = 0,1$, then there exists a scope δ, a thinning $\theta : \delta \sqsubseteq \gamma$, and thinnings $\phi_i : \gamma_i \sqsubseteq \delta$ such that $\phi_i\theta = \theta_i$ for $i = 0,1$ with the universal property that for any $\phi : \delta' \sqsubseteq \gamma$ and $\phi'_i : \gamma_i \sqsubseteq \delta'$ such that $\theta_i = \phi'_i\phi$, we have some $\theta' : \delta \sqsubseteq \delta'$ such that $\theta = \theta'\phi$ and $\phi'_i = \theta'\phi_i$ for $i = 0,1$. We define $\theta_0 \sqcup \theta_1 = \phi_0|\theta|\phi_1$, leaving δ implicit.

Proof
We shall see that θ is the pointwise disjunction of the θ_i, which needs must both γ_i embed in. More formally, we compute coproducts in the slice thus:

$$\begin{array}{rcll}
\varepsilon \sqcup \varepsilon & = & \varepsilon|\ \varepsilon\ |\varepsilon & \\
(\theta_0,0) \sqcup (\theta_1,0) & = & \phi_0|\theta,0|\phi_1 & \text{where } \phi_0|\theta|\phi_1 = \theta_0 \sqcup \theta_1 \\
(\theta_0,0) \sqcup (\theta_1,1) & = & \phi_0,0|\theta,1|\phi_1,1 & \text{where } \phi_0|\theta|\phi_1 = \theta_0 \sqcup \theta_1 \\
(\theta_0,1) \sqcup (\theta_1,0) & = & \phi_0,1|\theta,1|\phi_1,0 & \text{where } \phi_0|\theta|\phi_1 = \theta_0 \sqcup \theta_1 \\
(\theta_0,1) \sqcup (\theta_1,1) & = & \phi_0,1|\theta,1|\phi_1,1 & \text{where } \phi_0|\theta|\phi_1 = \theta_0 \sqcup \theta_1
\end{array}$$

The universal property holds by straightforward induction on the call graph of $\cdot \sqcup \cdot$. □

A direct consequence of the latter is that we can compute the *support* of a term — the smallest scope in which it is represented.

Lemma 76 (*support*)

For any $t : \mathsf{Term}\, l\, d\, \gamma$, there exist $\theta : \delta \sqsubseteq \gamma$ and $s : \mathsf{Term}\, l\, d\, \delta$ such that $t = s\theta$, with the universal property that whenever $t = s'\theta'$, there is some ϕ such that $\theta = \phi\theta'$. We may define $\lfloor t \rfloor = s|\theta$

Proof
Let us consider four cases. Firstly, the support of an atom is trivial.

$$\lfloor a \rfloor = a|\mathbf{0}$$

Secondly, for pairing (or radicals, or eliminations), we compute the pointwise disjunction of the supports of the components.

$$\lfloor (s\,.\,t) \rfloor = (s'\phi_0\,.\,t'\phi_1)|\theta \text{ where } \quad s', \theta_0 = \lfloor s \rfloor\, t', \theta_t = \lfloor t \rfloor\ = \lfloor \phi \rfloor_0 | \theta | \phi_1 = \theta_0 \sqcup \theta_1$$

Thirdly, variables have singleton support.

$$\lfloor x \rfloor = 1|x$$

Fourthly, the support of an abstraction requires us to separate the bound variable from the free variables.

$$\lfloor \backslash x\, t \rfloor = \backslash x\, t'(\mathbf{1} \otimes b)|\theta \text{ where} \lfloor t \rfloor = t'|\theta \otimes b$$

We may check that, indeed, $(\backslash x\, t'(\mathbf{1} \otimes b))\theta = \backslash x\, (t'(\mathbf{1} \otimes b)(\theta \otimes 1)) = \backslash x\, (t'(\theta \otimes b)) = \backslash x\, t$. The universal property of the support follows from that of $\cdot \sqcup \cdot$. $\square$

9 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 $? = t/(s : S)$.

10 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

$$\textsc{pre}\ \frac{T \rightsquigarrow T' \quad T' \ni t}{T \ni t} \qquad \textsc{post}\ \frac{e \in S \quad S \rightsquigarrow S'}{e \in S}$$

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.

11 Discussion

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

References

Asperti, Andrea, Ricciotti, Wilmer, Coen, Claudio Sacerdoti, & Tassi, Enrico. (2012). A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical methods in computer science*, **8**(1).

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.