The Types Who Say '\ni'

Conor McBride

September 5, 2017

1 Introduction

This paper documents a formalization of the basic metatheory for a bidirectional presentation of Martin-Löf's small and beautiful, but notoriously inconsistent dependent type theory from 1971 [Martin-Löf(1971)]. Perhaps more significantly, it introduces a methodology for constructing and validating bidirectional type systems, illustrated with a nontrivial running example. 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 and Turner(2000)].

2 The 1971 Rules

Let us first see the system which we are about to reorganise.

Really? Actually, I'm just guessing.

f, s, t, S, T	$::=$	$*$	the type of all types
	$\mid$	$(x : S) \to T[x]$	dependent function spaces
	$\mid$	$\lambda x : S.\ t[x]$	typed abstraction
	$\mid$	$f\ s$	application
	$\mid$	x	variable
Γ, Δ	$::=$	$\mathcal{E}$	empty context
	$\mid$	$\Gamma, x : S$	context extension, with freshly chosen x

It is my habit to be explicit (with square brackets) when introducing schematic variables in the scope of a binder: here, $T[x]$ and $t[x]$ may depend on the x bound just before, whereas the domain type S may not. It is, moreover, my habit to substitute such bound variables just by writing terms in the square brackets. For example, the β-contraction scheme is given thus:

$$(\lambda x : S.\ t[x])\ s \rightsquigarrow t[s]$$

The left-hand side is a *pattern*, which establishes schematic variables and makes clear their scope; the right-hand side is an *expression*, which must explain how the bound variable is instantiated.

Terms are identified up to α-conversion and substitution is capture-avoiding: the formalization uses a scope-safe de Bruijn index representation [de Bruijn(1972)].

Let us define $\cong$, 'β-convertibility', to be the equivalence and contextual closure of $\rightsquigarrow$. The typing rules will identify types up to $\cong$.

We have two judgment forms

context validity $\boxed{\Gamma \vdash \textsc{ok}}$ asserts that Γ is an assignment of types to distinct variables, where each type may depend on the variables given before;

type synthesis $\boxed{\Gamma \vdash t : T}$ asserts that the type T can be *synthesized* for the term t.

$\boxed{\Gamma \vdash \text{OK}}$

$$\frac{}{\mathcal{E} \vdash \text{OK}} \qquad \frac{\Gamma \vdash \text{OK} \quad \Gamma \vdash S \ : \ *}{\Gamma, x : S \vdash \text{OK}}$$

$\boxed{\Gamma \vdash t \ : \ T}$

$$\frac{\Gamma, x : S, \Delta \vdash \text{OK}}{\Gamma, x : S, \Delta \vdash x \ : \ S} \qquad \frac{\Gamma \vdash \text{OK}}{\Gamma \vdash * \ : \ *}$$

$$\frac{\Gamma \vdash S \ : \ * \quad \Gamma, x : S \vdash T[x] \ : \ *}{\Gamma \vdash (x : S) \to T[x] \ : \ *} \qquad \frac{\Gamma \vdash S \ : \ * \quad \Gamma, x : S \vdash t[x] \ : \ T[x]}{\Gamma \vdash \lambda x : S.\ t[x] \ : \ (x : S) \to T[x]}$$

$$\frac{\Gamma \vdash f \ : \ (x : S) \to T[x] \quad \Gamma \vdash s \ : \ S}{\Gamma \vdash f \ s \ : \ T[s]}$$

$$\frac{\Gamma \vdash t \ : \ S \quad \Gamma \vdash T \ : \ * \quad S \cong T}{\Gamma \vdash t \ : \ T}$$

I do not write explicit variable freshness requirements. Rather, I think of the turnstile as equipped with a supply of fresh names for free variables, while bound names are arbitrary. So, for example, in the rule for typing an abstraction, it is not that we hope for a coincidence of bound names but that we impose a standard choice of a free name when we extend the context.

The system has one rule for each syntactic construct and one rule (the 'conversion' rule) to impose the identification of types up to convertibility. If you look carefully at the rules for the syntax, you will see that the data left of the colon in the conclusion determine the data left of the colon in the premises; moreover, the data right of the colon in the premises determine the data right of the colon in the conclusion. That is to say that these five rules can be read as instructions for type synthesis. Only the conversion rule comes with no clear syntactic guidance: the essence of writing a type synthesis *algorithm* is to fix a particular strategy for deploying the conversion rule, then proving that strategy complete.

It is worth noting that the application rule has *two* occurrences of S right of the colon: implicitly, such a rule demands that two synthesized types agree precisely, but the conversion rule allows them to be brought into precise agreement by computation. Meanwhile, the conversion rule allows a type, once synthesized, to be modified by any amount of forward *or backward* computation. Backward creates an opportunity to introduce any old nonsense, as

$$(\lambda X : *.\ *)\ (* * * *) \rightsquigarrow *$$

To prevent infection with such nonsense, the conversion rule insists that we check we end up with a valid type. Now, as it happens, our reduction system is confluent and moreover, forward computation preserves type. As a result, if we know that $S \cong T$ are valid types, then they have a common reduct R: we can compute S to R and T to R without stepping outside the valid types at any point. Hence, the conversion rule's check that T is a type is both necessary and sufficient.

A further point of note is that the type synthesis rules have no axioms. The *only* axiom is that the empty context is uncontroversially valid. The two 'base cases' of typing, for $*$ and for variables, have premises ensuring context validity. The following 'sanity clauses' are admissible:

$$\frac{\Gamma \vdash t \ : \ T}{\Gamma \vdash \text{OK}} \qquad \frac{\Gamma \vdash t \ : \ T}{\Gamma \vdash T \ : \ *}$$

You can see that both of the rules which extend the context directly check the validity condition for so doing: the type synthesis rule for abstraction makes crucial use of the type annotation in $\lambda x : S.\ t[x]$, without which it would be necessary to guess the type of x from its uses. The type of the abstraction body, being generic in the argument, allows us to form the correct function type unproblematically. Meanwhile, to see why synthesized types are well formed (for application in particular), we need stability of typing under substitution, which is as much as to say that we can substitute a (suitably weakened) typing derivation for some $s : S$ in place of all uses of the variable rule which witness $x : S$. Stability of typing under substitution relies, of course, on stability of

computation under substitution. However, our computation rule never makes any requirements about the presence of free variables, matching only syntactic constructs which are preserved by substitution, so it would be quite a surprise if stability under substitution were to fail.

The rule for $*$, often called 'Type-in-Type', opens the door to paradox. Famously, Girard had shown that the Burali-Forti paradox could be embedded in System U, which has two impredicative layers. Martin-Löf's system offered arbitrary impredicativity, making it easy to embed System U. However, despite being inconsistent and non-normalizing, this theory does enjoy the basic type preservation and progress properties we expect of functional programming languages, and many of the type theories we have today are effectively refinements with sufficient paranoia to prevent loops.

Stick in the Hurkens construction?

3 The Bidirectional Syntax

The idea behind bidirectional type systems is to make use of the way we sometimes know type information in advance. If we start from a type, there may be fewer choices to determine an inhabiting term. The type represents a *requirement*, rather than a *measurement*. We work a little more precisely at managing the flow of type information, and we gain some convenience and cleanliness. I like to start by separating the syntactic categories into checkable *constructions* and synthesizable *eliminations*.

s, t, S, T	$::=$	$*$	the type of all types
	$\mid$	$(x : S) \to T[x]$	dependent function spaces
	$\mid$	$\lambda x.\ t[x]$	untyped abstraction
	$\mid$	$\underline{e}$	embedded elimination
e, f	$::=$	x	variable
	$\mid$	$f\ s$	application
	$\mid$	$\vdots$	to be continued...

I have omitted one elimination form by way of generating a little suspense: let us see how we get along without it. Type formation and value introduction syntax sits on the *construction* side; variable usage and application sit on the *elimination* side. Eliminations embed into constructions, with a relatively unobtrusive but nontrivial underline: in a real implementation, there is no need to spend characters on this feature, but when studying metatheory, it helps to see where it is used. The reverse embedding is *not* available, and we shall see why when we study the rules.

At this stage, however, it is worth noting the following:

- the syntax forbids the expression of β-redexes;
- every elimination has a variable at its head, with a spine of arguments;
- it is syntactically invalid to substitute a construction for a variable.

Let us formalize this syntax in Agda. We may enumerate the choice of syntactic category, or Sort (in its universal algebra sense, rather than its 'type of types' sense) for short.

```
data Sort : Set where chk syn : Sort
```

The constructor names highlight the distinction between checking and synthesis. Variables must be in *scope*. In more complex syntaxes, a scope is a list of variable Sorts, but here we have variables only of sort syn, so a Natural number suffices to represent a valid scope. [**Agda.** The BUILTIN pragma instructs Agda to allow us decimal numerals for numbers.]

```
data Nat : Set where ze : Nat; su : Nat → Nat
 {-# BUILTIN NATURAL Nat #-}
```

A syntactically valid term has a scope and a sort.

```
data Tm (n : Nat) : Sort → Set where                                  -- informally...
  *   :                                                  Tm n chk     -- *
  Π   : (S : Tm n chk) (T : Tm (su n) chk)          →    Tm n chk     -- (x:S) → T[x]
  λ   :                (t : Tm (su n) chk)          →    Tm n chk     -- λx. t[x]
  ϵ   : (e : Tm n syn)                              →    Tm n chk     -- e
  #   : (i : Var n)                                 →    Tm n syn     -- x
  _$_ : (f : Tm n syn) (s : Tm n chk)               →    Tm n syn     -- f s
```

[**Agda.** In the declaration of Tm, we see $(n : \mathsf{Nat})$ abstracted left of :, scoping over the whole declaration. Some young people might incorrectly refer to n as a 'parameter' of Tm, but it is not a parameter in Dybjer's definitive sense [**?**]: Tm's first argument varies in recursive positions, so we are taking a least fixpoint on $\mathsf{Nat} \to \mathsf{Sort} \to \mathsf{Set}$, not a number of fixpoints on $\mathsf{Sort} \to \mathsf{Set}$. The n is thus an *index*, but one in which the constructor return types are *uniform*. Hancock calls such indices 'protestant' to distinguish them from the 'catholic' indices (such as our Sort) which may be individually instantiated in constructor return types, as paradigmatically done in the inductively defined equality type, enabling the miracle of transubstantiation.]

It has always struck me as an intensely frustrating business, defining an inductive datatype to represent some syntax of terms, and then showing that it really works like a syntax by defining operations such as substitution and proving that they exhibit the appropriate structure. When appropriately viewed, these types are syntactic by construction. Sadly, the appropriate view is not the machine's view: the business of teaching the machine to see syntax is for another time.

Back to our datatype, note that the types of Π and λ expose their variable binding power by incrementing the scope count for the range of function types and the body of an abstraction. The Var n type represents a choice of one variable from the n available. I shall resolve the mystery of its definition directly.

4 Variables as a Line in Pascal's Triangle

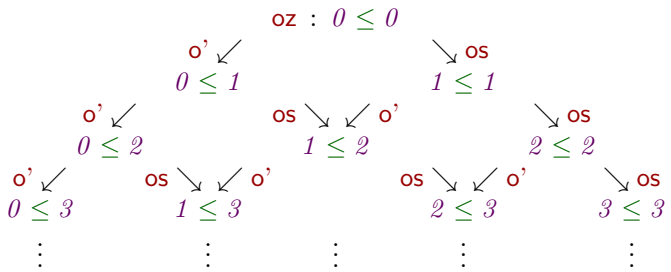
Choosing *one* variable from those in scope is the unitary case of choosing *some* variables from those in scope. Choosing *some* variables amounts to constructing an *order-preserving embedding* (or a 'thinning', for short) from the chosen variables into all the variables. Such a choice is possible only if we have enough variables, so we acquire a proof-relevant version, $\leq$, of the 'less or equal' relation, preserved by the constructors of numbers (as shown by oz and os), but allowing us to omit a 'target' variable (with o') whenever we choose, or rather, whenever we choose-not.

With an eye to the future, I introduce a more general notion of *scope morphism*, which allows us to embed variables from source to target scope, but also to map them to some other sort of image, X. We acquire $\leq$ as the special case where X is empty.

```
data Mor (X : Nat → Set) : Nat → Nat → Set where
  oz :                                    Mor X  ze     ze
  os : ∀ {n m} →  Mor X n m →             Mor X (su n) (su m)
  o' : ∀ {n m} →  Mor X n m →             Mor X      n ( su m)
  ox : ∀ {n m} →  Mor X n m → X m →       Mor X (su n)     m
data Zero : Set where
_≤_ : Nat → Nat → Set
_≤_ = Mor (λ _ → Zero)
Var : Nat → Set
Var = (1 ≤_)
```

Our Var is then given as an operator section, fixing the number of variables to choose as 1.

We may tabulate these $\leq$ types as a version of Pascal's Triangle, replacing the *number* of paths to a given position by the *type* of the embeddings they generate:



The left spine of the triangle (usually all 1s) gives the unique embedding from the empty scope to any scope. Meanwhile, the right spine gives the identity embedding. In fact, we can and should construct these operations for scope morphisms in general.

```
oe : ∀ {n X} → Mor X ze n
oe {ze}   = oz
oe {su n} = o' (oe {n})
```

```
oi : ∀ {n X} → Mor X n n
oi {ze}   = oz
oi {su n} = os (oi {n})
```

[**Agda.** Curly braces in function types mark arguments to be suppressed by default. Curly braces in applications and abstractions override default suppression, as is necessary here for purposes of pattern matching. That is, in a most unmilnerian manner, arguments usually delivered by "type inference" have relevance in execution. We could perfectly well have written o' oe and os oi on the right of the above step cases, suppressing the {n}s. I give them explicitly only to make legible the structural justification for the recursion.]

Var is the next south-westerly diagonal down from the left spine, $1 \leq n$, where each type has size n and may be taken to represent the choice of one variable from n in scope. The ith de Bruijn index (counting from 0) is given as o'^{i} (os oe). I use $(1 \leq \_)$ rather than the more traditional 'Fin' family to emphasize that variable sets arise from $\leq$, the category of thinnings. We have the identity thinning, but you will have to wait for the composition. Why? Because we shall define compositions for Mor in general, not just for $\leq$, and that will demand additional equipment.

5 Action and Composition of Scope Morphisms

In any case, we must consider how to make our morphisms act on *terms*.

```
module ACT (X : Nat → Set)
    -- operations on X will appear as we discover what we need
where

_/_ : ∀ {n m sort} → Tm n sort → Mor X n m → Tm m sort

# i / σ = ?

*       / σ = *
Π S T   / σ = Π (S / σ) (T / os σ)
λ t     / σ = λ (t / os σ)
ϵ e     / σ = ϵ (e / σ)
(f $ s) / σ = (f / σ) $ (s / σ)
```

We can implement the (functorial, as we shall see) action of a morphism for everything but variables, using the (functorial, as we shall see) os constructor.

Let us think about what the morphism σ might do to some source variable i. It will either be sent to some target variable by os or mapped to some X by an ox. Correspondingly, we should be

able to implement some operation

```
fetch : ∀ {n m} → Var n → Mor X n m → Var m + X m
```

where $+$ is the usual sum type, $+_e$ its case analysis operator, and $+$ its functorial action.

```
data _+_ (S T : Set) : Set where
  inl : S → S + T
  inr : T → S + T
_+e_ : {S T U : Set} → (S → U) → (T → U) → (S + T) → U
(f +e g) (inl s) = f s
(f +e g) (inr t) = g t
_+_ : {S S' T T' : Set} → (S → S') → (T → T') → (S + T) → (S' + T')
f + g = (inl ∘ f) +e (inr ∘ g)
```

We can then complete our action, provided we know how to turn an X into an elimination. If we add a parameter to the ACT module,

```
(tmX : ∀ {n} → X n → Tm n syn)
```

then we can make some progress.

```
fetched : ∀ {m} → Var m + X m → Tm m syn
fetched = # +e tmX


# i / σ = fetched (fetch i σ)
```

However, to finish the job we must define fetch. This will clearly require us to be shift Xs under binders. ACT needs a further module parameter

```
(wkX : ∀ {n} → X n → X (su n))
```

so that we may complete the construction:

```
fetch i       (o' σ)    = (o' + wkX) (fetch i σ)
fetch (ox i ()) σ
fetch (os i)  (os σ)    = inl (os oe)
fetch (o' i)  (os σ)    = (o' + wkX) (fetch i σ)
fetch (os i)  (ox σ x)  = inr x
fetch (o' i)  (ox σ x)  = fetch i σ
```

We immediately acquire the action of thinnings: as X is empty, tmX and wkX are vacuous.

```
_/t_ : ∀ {n m sort} → Tm n sort → n ≤ m → Tm m sort
_/t_ = ACT._/_ (λ _ → Zero) (λ ()) (λ ())
```

[**Agda.** Unlike in Haskell, where () means the boring value in the unit type, Agda uses () for the *impossible* pattern in an empty type. Moreover, when you point out that an input is impossible, you are absolved of the responsibility to explain what to do with it. Correspondingly $\lambda\,()$ is just the function we need to map non-existent Xs into anything.]

Now that we can thin, *substitution* is easy, too. We may now permit morphisms which turn variables into eliminations.

```
Elim : Nat → Set
Elim n = Tm n syn
```

To give the action of a substitution, we need to embed Elim into itself, which is the identity, and to weaken Elim, which we do with the thinning o' oi, 'missing' the top variable.

```
_⇒_ : Nat → Nat → Set
_⇒_ = Mor Elim
_/ₛ_ : ∀ {n m sort} → Tm n sort → n ⇒ m → Tm m sort
_/ₛ_ = ACT._/_ Elim id (_/ₜ o' oi)
```

We can play the same sort of game when defining composition.

```
module COMP (S T U : Nat → Set)
            (front : ∀ {n m} → S n → Mor T n m → U m)
            (back  : ∀ {m} →          T m →      U m)
  where
    _⨟_ : ∀ {p n m} → Mor S p n → Mor T n m → Mor U p m
    σ        ⨟ o' τ    = o' (σ ⨟ τ)
    ox σ s   ⨟ τ       = ox (σ ⨟ τ) (front s τ)
    os σ     ⨟ ox τ t  = ox (σ ⨟ τ) (back t)
    o' σ     ⨟ ox τ t  = σ ⨟ τ
    os σ     ⨟ os τ    = os (σ ⨟ τ)
    o' σ     ⨟ os τ    = o' (σ ⨟ τ)
    oz       ⨟ oz      = oz
```

[**Agda.** I have been careful to order the lines of this definition so that it is lazy in its front argument whenever its back is o' τ. Agda's translation from pattern matching to case analysis trees is quite naïve: the fact that the first line does not split σ but does match the back argument is what ensures this laziness. The first line takes priority over the second: they do overlap. Shifting the first two lines later in the definition while retaining their order would retain the extensional properties of the function, but change the intensional properties.]

As we try to implement composition, we discover that we need the means to transport S and T values (packed by ox) across to U. For thinning, of course, these are vacuous requirements.

```
_⨟ₜ_ : ∀ {p n m} → p ≤ n → n ≤ m → p ≤ m
_⨟ₜ_ = COMP._⨟_ (λ _ → Zero) (λ _ → Zero) (λ _ → Zero)
                (λ ()) (λ ())
```

For substitutions, we need merely know how they act.

```
_⨟ₛ_ : ∀ {p n m} → p ⇒ n → n ⇒ m → p ⇒ m
_⨟ₛ_ = COMP._⨟_ Elim Elim Elim _/ₛ_ (λ t → t)
```

Now, let us at least state some laws in terms of the equality type:

```
data _≃_ {l} {X : Set l} (x : X) : X → Set l where refl : x ≃ x
```

I omit the unedifying proofs that show thinnings and substitutions both form categories acting functorially on terms.

$$
\begin{array}{lll}
 & (t \mathbin{/_\mathsf{t}} \mathsf{oi}) \simeq t & ((t \mathbin{/_\mathsf{t}} \theta_0) \mathbin{/_\mathsf{t}} \theta_1) \simeq (t \mathbin{/_\mathsf{t}} (\theta_0 \mathbin{⨟_\mathsf{t}} \theta_1)) \\
(\mathsf{oi} \mathbin{⨟_\mathsf{t}} \theta) \simeq \theta & (\theta \mathbin{⨟_\mathsf{t}} \mathsf{oi}) \simeq \theta & ((\theta_0 \mathbin{⨟_\mathsf{t}} \theta_1) \mathbin{⨟_\mathsf{t}} \theta_2) \simeq (\theta_0 \mathbin{⨟_\mathsf{t}} (\theta_1 \mathbin{⨟_\mathsf{t}} \theta_2)) \\
 & (t \mathbin{/_\mathsf{s}} \mathsf{oi}) \simeq t & ((t \mathbin{/_\mathsf{t}} \sigma_0) \mathbin{/_\mathsf{t}} \sigma_1) \simeq (t \mathbin{/_\mathsf{s}} (\sigma_0 \mathbin{⨟} \sigma_1)) \\
(\mathsf{oi} \mathbin{⨟_\mathsf{s}} \sigma) \simeq \sigma & (\sigma \mathbin{⨟_\mathsf{s}} \mathsf{oi}) \simeq \sigma & ((\sigma_0 \mathbin{⨟_\mathsf{s}} \sigma_1) \mathbin{⨟_\mathsf{s}} \sigma_2) \simeq (\sigma_0 \mathbin{⨟_\mathsf{s}} (\sigma_1 \mathbin{⨟_\mathsf{s}} \sigma_2))
\end{array}
$$

The proofs are implemented in the same style as the programs: generic results are proved with conditions, then instantiated trivially for thinnings to obtain the lemmas needed to show that the conditions hold for substitutions.

It is worth noting that construction of $/_\mathsf{t}$ and $/_\mathsf{s}$, together with the proofs of the laws they satisfy, is entirely generic for syntaxes with binding. Really, we should present all the syntaxes as a *universe* and do the construction once.

With our basic syntactic machinery now in place, let us return to designing the type system.

6 Type-*has*-Type

We have two syntactic categories, so we need at least two judgment forms:

type checking $\Gamma \vdash T \ni t$ constructions are checked with respect to a given type;

type synthesis $\Gamma \vdash e \in S$ eliminations have their types synthesized, from the type of their head variable, which is given in the context.

The 'forward' $\in$ of type synthesis is pronounced "in", with LaTeX macro `\in`, or perhaps "is a(n)" or "inhabits"; its reverse, used for checking, may be pronounced "ni", for its LaTeX macro is `\ni`, but might be more intelligibly pronounced "has" or "accepts".

Many other authors keep terms to the left of types and use arrows (directions vary) to make the checking/synthesis distinction. I insist on retaining the left-to-right flow of *time* through the syntax of judgments, which means that when checking, the type must come before the term.

In fact, I classify the schematic positions in judgment forms as having one of three *modes*:

inputs are given in advance and *required* to be valid (in some sense which should be specified);

subjects are the things under scrutiny, whose validity is the question;

outputs are data synthesized in the validation process and *guaranteed* to be valid (in some sense which should be clearly specified).

For the above judgment forms, we shall have

$$\underset{\text{input}}{\Gamma} \quad \vdash \quad \underset{\text{input}}{T} \quad \ni \quad \underset{\text{subject}}{t} \qquad\qquad \underset{\text{input}}{\Gamma} \quad \vdash \quad \underset{\text{subject}}{e} \quad \in \quad \underset{\text{output}}{S}$$

In order to specify the requirements and guarantees (but not to give the rules themselves), we shall also need a context validity judgment, $\Gamma \vdash \textsc{ok}$, for which Γ is considered the subject. We should expect every judgment input to have a requirement for which it is the subject and every judgment output to have a guarantee for which it is the subject. Here,

$$\begin{array}{lrl} \Gamma \vdash T \ni t & \text{requires} & \Gamma \vdash \textsc{ok} \\ & \text{requires} & \Gamma \vdash * \ni T \\ \\ \Gamma \vdash e \in S & \text{requires} & \Gamma \vdash \textsc{ok} \\ & \text{guarantees} & \Gamma \vdash * \ni S \end{array}$$

and we are correspondingly not free to write down any old rubbish by way of typing rules. Each rule gives rise to proof obligations which we must check. However, in the rule to establish a particular judgment, the requirements even to propose the judgment are *presumed*, not revalidated: as it were, "We would not be asking this question if we did not already know so-and-so.".

There is more to say about the impact of *mode* on valid notions of typing rule.

1. The inputs of conclusions and the outputs of premises must be *patterns*, which may match against any construct of the calculus *except free variables* and are the binding sites for the schematic variables of the rules.

2. The outputs of conclusions and the inputs of premises must be *expressions*, making use of the schematic variables in scope and instantiating any bound variables they may have. Scope flows clockwise round the rules, starting from the inputs of the conclusion, accumulating left-to-right through the premises, finishing with the outputs of the conclusion.

3. Only the schematic variables in the conclusion's *subjects* are in scope for the subjects of the premises, and they must all occur in exactly one premise.

4. A schematic subject variable becomes in scope for *expressions* only after it has been the subject of a premise (and thus achieved some measure of trust).

These four conditions form the basis of a kind of 'religion' of typing rules: let us obey them for now and consider breaking them when we are older and more aware of the consequences of our actions.

The type checking and context validity rules are as follows:

$$\boxed{\Gamma \vdash T \ni t} \qquad \frac{}{\Gamma \vdash * \ni *} \qquad \frac{\Gamma \vdash * \ni S \quad \Gamma, x{:}S \vdash * \ni T[x]}{\Gamma \vdash * \ni (x{:}S) \to T[x]} \qquad \frac{\Gamma, x{:}S \vdash T[x] \ni t[x]}{\Gamma \vdash (x{:}S) \to T[x] \ni \lambda x.\, t[x]}$$

$$\frac{\Gamma \vdash e \in S \quad S \equiv T}{\Gamma \vdash T \ni \underline{e}}$$

$$\boxed{\Gamma \vdash \textsc{ok}} \qquad \frac{}{\mathcal{E} \vdash \textsc{ok}} \qquad \frac{\Gamma \vdash \textsc{ok} \quad \Gamma \vdash * \ni S}{\Gamma, x{:}S \vdash \textsc{ok}}$$

More rules will follow. For now, we start with 'Type-has-Type', without revalidating the context (for we *presume* its validity, given that the context is an input to the conclusion). Note that it is not only the case that our entitlements *allow* us to omit a $\Gamma \vdash \textsc{ok}$ premise, but also the case that our religion *forbids* such a premise, for it would have Γ as its subject, and Γ is an *input* variable, not in scope for the subjects of premises. In time, this will save our bacon.

Meanwhile, to check a function type, we check its components: once the domain has been checked, we may extend the context (maintaining its validity) and check the range. To check an abstraction, we presume that the function type is indeed a type, and hence by inversion that its domain and range are types in the relevant contexts: we may thus proceed directly to checking that the range type accepts the body of the untyped abstraction, using the input domain as the type of the bound variable. Finally, to *check* an embedded elimination, we first *synthesize* the type of the elimination, and then insist that the two candidate types match *exactly* (i.e., that they are α-equivalent, which is just syntactic identity for a de Bruijn representation), rather than up to some (so far unspecified, in any case) notion of conversion.

Now, some of you may wonder why we do not synthesize the types for $*$ and $(x{:}S) \to T[x]$, given that they clearly inhabit $*$ if they inhabit anything at all. While that works for this system with one universe, it is unstable with respect to overloading: type checking allows us to overload introduction forms for distinct types, including the overloading of type formation constructs within different universes. Such overloading rules out type synthesis but has no impact on type checking. The fix to restore normalization exactly requires us to introduce multiple universes and overload the function type constructor, so let us stick with type checking for these things.

Meanwhile, the heart of type synthesis is the variable rule, extracting the type of the head of an elimination from the context.

$$\frac{}{\Gamma, x{:}S, \Delta \vdash x \in S}$$

The synthesized type comes directly from the input context which is presumed valid, and must thus confirm that S is indeed a type, which is what we must guarantee.

For application, we *synthesize* the type of the function (which is being eliminated), then *check* that it is being eliminated in a manner consistent with its type. We can get most of the way round before we run into trouble:

$$\frac{\Gamma \vdash f \in (x{:}S) \to T[x] \quad \Gamma \vdash S \ni s}{\Gamma \vdash f\ s \in ?}$$

Synthesizing the type (and we are guaranteed that it is a type) of the function, we may extract the (by inversion also a type) domain and use it to check the argument. However, when we come to give the output type of the whole application, the wheel comes off. We may be sure that $\Gamma, x{:}S \vdash * \ni T[x]$, and we surely want to give a type by choosing a suitable replacement for that x, but we may not give $T[s]$, as s is not in the same syntactic category as x. Indeed, substituting such an s for x might create β-redexes which we have thus far excluded.

One possibility is to seek a derived notion of *hereditary* substitution [Watkins et al.(2003)Watkins, Cervesato, Pfenning which computes out any such β-redexes on the fly, restoring syntactic legitimacy. However, we

know that any such operation will not be well defined: it can be persuaded to require the normalization of anything, and this language is surely non-normalizing. We might deal with the partiality of hereditary substitution by defining it *relationally*, but that is only to postpone the problem: in a non-normalizing calculus, hereditary substitution is not stable under hereditary substitution, as we can substitute an inert variable with just the term required to kick off an infinite computation. Correspondingly, the key stability property that drives the proof of type preservation will fail.

We had better think it out again.

7 Radicals Recover Small-Step Behaviour

When we come to instantiate $T[x]$ with s, we do at least know S, the type of s. We can reacquire the ability to substitute if we extend the language of *eliminations* with type-annotated terms, which I call *radicals* by analogy with the chemical notion.

$$e, f ::= \ldots \mid t{:}T$$

We should also add radicals to the declaration of Tm and extend $/$ accordingly.

$$\_{:}\_ \;:\; \mathsf{Tm}\ n\ \mathsf{chk} \;\to\; \mathsf{Tm}\ n\ \mathsf{chk} \;\to\; \mathsf{Tm}\ n\ \mathsf{syn}$$

$$(t : T) \,/\, \sigma \;=\; (t \,/\, \sigma) : (T \,/\, \sigma)$$

The associated typing rule allows us to change direction in the other direction from embedding, at the cost of making the intermediate type explicit: the type we synthesize is exactly the type we supply.

$$\frac{\Gamma \vdash * \ni T \quad \Gamma \vdash T \ni t}{\Gamma \vdash t{:}T \in T}$$

We may now complete the application rule:

$$\frac{\Gamma \vdash f \in (x{:}S) \to T[x] \quad \Gamma \vdash S \ni s}{\Gamma \vdash f\ s \in T[s{:}S]}$$

However, we have opened a further door at the same time. A radical can stand at the head of an elimination, taking its type not from the context but from its own explicit annotation. In particular, we can form something which looks a touch like a β-redex:

$$(\lambda x.\ t[x] : (x{:}S) \to T[x])\ s$$

Inversion tells us that for this to be well typed in some context Γ, we must know the following:

$$\Gamma \vdash * \ni S \qquad \Gamma, x{:}S \vdash * \ni T[x] \qquad \Gamma, x{:}S \vdash T[x] \ni t[x] \qquad \Gamma \vdash S \ni s$$

and have synthesized the type $T[s{:}S]$ for the whole application We can thus give this would-be β-redex a computational behaviour:

$$(\lambda x.\ t[x] : (x{:}S) \to T[x])\ s \;\rightsquigarrow_\beta\; t[s{:}S] : T[s{:}S]$$

Note that, on the one hand, the type annotation on the contractum is essential (for the redex is an elimination, hence so must be its contractum), but on the other hand, the type annotation is helpful: a radical at a function type has 'reacted' to release radicals at both domain and range types (which might be function types, causing further computation). It is easy to check (once we have established stability under substitution), that

$$\Gamma \vdash * \ni T[s{:}S] \qquad \Gamma \vdash T[s{:}S] \ni t[s{:}S]$$

and hence the reduct gives us the same type as the contractum.

However, as soon as a radical is being embedded rather than eliminated, its computational role is finished, so the type annotation is no longer required. We acquire the υ-reduction scheme

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

That is, an introduction form is activated by annotation with its type, allowing its elimination form to compute, then eventually (we hope) deactivated once all the eliminations in its spine have been completed. In a *predicative* system, the types get visibly smaller (in some suitably ramified sense) at each step. In our impredicative, indeed inconsistent system, we have no such guarantee. However, we must have checked (in the relevant context Γ) that $\Gamma \vdash * \ni T$ $\Gamma \vdash T \ni t$, allowing us to check both

$$\Gamma \vdash T \ni \underline{t{:}T} \qquad \Gamma \vdash T \ni t$$

again suggesting we have some chance of achieving type preservation.

But there's a but. Now that we have introduced a small-step computation system, we must ensure that typing respects it. Let us write an unlabelled $\rightsquigarrow$ for small-step reduction—the closure of either contraction scheme under all contexts—and add the *pre-* and *post-computation* rules, respectively,

$$\frac{T \rightsquigarrow T' \quad \Gamma \vdash T' \ni t}{\Gamma \vdash T \ni t} \qquad \frac{\Gamma \vdash e \in S \quad S \rightsquigarrow S'}{\Gamma \vdash e \in S'}$$

These rules obey the requirements of our mode-religion if we consider reduction to be moded

$$\text{input} \rightsquigarrow \text{output}$$

even though reduction is not deterministic. There is often a choice as to which redex to contract.

I have chosen the orientation of these rules carefully. We may precompute a type before checking it, e.g., ensuring that a type has the form of a function type before checking an abstraction; we may post-compute the type of a function being applied, so that it reduces to the form of a function type, allowing us to check the argument.

If you are paying attention, you will have spotted a catch or two. These two rules, governing the interaction of computation and types, are not syntax-directed, just as the conversion rule was not in the 1971 system. They can be invoked anytime, which also seems to derail or at least complicate the informal appeals to inversion in the arguments above. We shall need to work a little to make sure we have not introduced a problem as we try to establish type preservation. **Can we show systematically that it is enough to establish type preservation for the contraction schemes in the absence pre and post, by making use of confluence and modes? I suspect so.**

The other catch is that we acquire some type annotation flotsam in our normal forms. The annotation in

$$(\underline{y} : (x{:}S) \to T[x])\ s$$

will not compute away either by β or by υ. We might consider simplifying $\underline{e} : T$ to e, but that would give us nasty critical pairs; we might insist that such an e have a free variable at its head, but that property is not stable under substitution. We shall do neither of these things and yet come to no harm. However, we might find it frustrating if such flotsam makes types incompatible, so let us return to this question later.

One positive, however, is that there is no longer any *backward* computation in our calculus. The anarchic 1971 conversion rule has become the rather more controlled observation that the following is now admissible (by recursion on reduction sequences):

$$\frac{\Gamma \vdash e \in S \quad S \rightsquigarrow^* R \quad T \rightsquigarrow^* R}{\Gamma \vdash T \ni \underline{e}}$$

A synthesized type is good for a checked type if they have a common reduct, which is just as good as convertibility in a confluent system.

The bidirectional system offers us fewer opportunities to deploy convertibility as opposed to reduction. Here, we rely on a crucial property brought to prominence by Geuvers [**?**]. If a term is *convertible* with a canonical form (with a given head constructor and whatever substructures), then it *reduces* to a canonical form (with that constructor and convertible corresponding substructures). That is, reduction is good for conversion when seeking to establish that a type is $*$, or that it is $(x : S) \to T[x]$, as happens both when checking types for introductions and when synthesizing types for eliminations.

8 Formalizing Reduction

To formalize the typing rules, we must first formalize reduction. We may give an inductive family capturing exactly *one-step* reduction, with a base constructor for each contraction scheme and step constructors for each possible contextual closure.

```
data _⇝_ {n : Nat} : ∀ {sort} → Tm n sort → Tm n sort → Set where
  β    : ∀ {t T s S} →
         ((λ t ∶ Π S T) $ s) ⇝ ((t ∶ T) /s (ox oi (s ∶ S)))
  υ    : ∀ {t T} →
         ϵ (t ∶ T) ⇝ t
  Π_L  : ∀ {S S′ T} → S ⇝ S′ → Π S T ⇝ Π S′ T
  Π_R  : ∀ {S T T′} → T ⇝ T′ → Π S T ⇝ Π S T′
  λ    : ∀ {t t′} → t ⇝ t′ → λ t ⇝ λ t′
  ϵ    : ∀ {e e′} → e ⇝ e′ → ϵ e ⇝ ϵ e′
  ap_L : ∀ {f f′ s} → f ⇝ f′ → (f $ s) ⇝ (f′ $ s)
  ap_R : ∀ {f s s′} → s ⇝ s′ → (f $ s) ⇝ (f $ s′)
  ra_L : ∀ {t t′ T} → t ⇝ t′ → (t ∶ T) ⇝ (t′ ∶ T)
  ra_R : ∀ {t T T′} → T ⇝ T′ → (t ∶ T) ⇝ (t ∶ T′)
```

A Syntax, Formally

There's an inevitable dilemma about whether to do this explicitly in Agda.

When we model a syntax, an 'object language', its terms can be classified into object *sorts*, ι. (Sometimes, we may even identify the notion of 'sort' with the object language's notion of 'type', but that is far from crucial.) The object sorts for our language are constructions and eliminations.

$$\iota \quad ::= \quad \mathsf{cn} \quad | \quad \mathsf{el}$$

To account for variable binding, we construct a *meta*-level notion of *kind*, κ, by closing sorts under what might look to the casual observer like a function space, but is most emphatically weaker than that.

$$\kappa \quad ::= \quad \iota \quad | \quad [\kappa]\kappa'$$

The kind $[\kappa]\kappa'$ means 'terms in κ which bind a variable of kind κ', not 'functions from terms of kind κ to terms of kind κ''. The only functions thus represented are those which *substitute* the bound variable: functions which do the common functional thing of inspecting their input are absolutely ruled out.

For each *sort*, ι, we must give the *signature*, $\Sigma(\iota)$, of its constructs, specifying the *kind* of each subterm. Here, we may give this translation of our earlier grammar, omitting variables (which are not a notion specific to this theory), and giving explicit prefix forms for the remainder.

$$\begin{array}{rcl}
\Sigma(\mathsf{cn}) & ::= & * \\
& | & \Pi\ \mathsf{cn}\ [\mathsf{el}]\mathsf{cn} \\
& | & \lambda\ [\mathsf{el}]\mathsf{cn} \\
& | & \upsilon\ \mathsf{el} \\
\Sigma(\mathsf{el}) & ::= & \alpha\ \mathsf{el}\ \mathsf{cn} \\
& | & \epsilon\ \mathsf{cn}\ \mathsf{cn}
\end{array}$$

For any kind-indexed family of sets $T(\kappa)$ (for 'term'), we may give the set of constructions, $\Sigma^{\mathsf{c}}(T, \iota)$

$$\Sigma^{\mathsf{c}}(T, \iota) = \{c\ \vec{t} \mid c\ \vec{\kappa} \in \Sigma(\iota), \forall i.\, t_i \in T(\kappa_i)\}$$

These are prefix-Polish terms with Ts as subterms. Our construction of terms will instantiate T recursively, 'tying the knot'. We should note that $\Sigma^{\mathsf{c}}(-, \iota)$ is a functor from the category of kind-indexed sets with kind-preserving functions to the category of sets, acting structurally on the subterms.

Now let us define scoped and kinded syntax.

A.1 The Category of Scopes and Thinnings

A *scope* is a list of kinds, growing on the right:

$$\gamma, \delta, \zeta \quad ::= \quad \mathcal{E} \quad | \quad \gamma, \kappa$$

It will often be convenient to give kinds in *spine* form, $[\delta]\iota$, loading all the bound variable kinds into a scope, leaving an object sort. These scopes form the objects of a category.

The morphisms of this category are the *thinnings* (θ, ϕ) between scopes, $\gamma \leq \delta$:

$$\frac{}{\mathcal{E} \in \mathcal{E} \leq \mathcal{E}} \qquad \frac{\theta \in \gamma \leq \delta}{\theta, \kappa \in (\gamma, \kappa) \leq (\delta, \kappa)} \qquad \frac{\theta \in \gamma \leq \delta}{\theta^{\dagger}_{\kappa} \in \gamma \leq (\delta, \kappa)}$$

By careful choice of notation, we have identity thinnings.

$$\gamma in \gamma \leq \gamma$$

Composition of thinnings (which I write diagramatically) is definable:

$$\frac{\theta \in \gamma_0 \leq \gamma_1 \quad \phi \in \gamma_1 \leq \gamma_2}{\theta;\phi \in \gamma_0 \leq \gamma_2}$$

$$\begin{array}{rcl} \mathcal{E};\mathcal{E} & = & \mathcal{E} \\ (\theta,\kappa);(\phi,\kappa) & = & (\theta;\phi),\kappa \\ (\theta\dagger);(\phi,\kappa) & = & (\theta;\phi)\dagger \\ \theta;(\phi\dagger) & = & (\theta;\phi)\dagger \end{array}$$

It is easy to check that the categorical laws hold: if $\theta \in \gamma \leq \delta$, then

$$\gamma;\theta \;=\; \theta \;=\; \theta;\delta$$

and

$$(\theta_0;\theta_1);\theta_2 \;=\; \theta_0;(\theta_1;\theta_2)$$

Moreover, our definitions have ensured that $-,\kappa$, acting as it does both on scopes and on thinnings between those scopes, is a *functor*.

Every thinning $\theta \in \gamma \leq \delta$ induces a complementary scope $\overline{\theta}$ and a complementary thinning $\widehat{\theta} \in \overline{\theta} \leq \delta$ inverting the selection of scope items included by θ:

$$\begin{array}{rcl} \overline{\mathcal{E}} & = & \mathcal{E} \\ \overline{\gamma,\kappa} & = & \overline{\gamma} \\ \overline{\gamma\dagger_\kappa} & = & \overline{\gamma},\kappa \end{array} \qquad \begin{array}{rcl} \widehat{\mathcal{E}} & = & \mathcal{E} \\ \widehat{\gamma,\kappa} & = & \widehat{\gamma}\dagger_\kappa \\ \widehat{\gamma\dagger_\kappa} & = & \widehat{\gamma},\kappa \end{array}$$

Note that $\widehat{\gamma} \in \mathcal{E} \leq \gamma$ makes $\mathcal{E}$ the initial object in the category.

Let us write $\kappa \leftarrow \gamma$ for the set of *singleton* thinnings $(\mathcal{E},\kappa) \leq \gamma$. These look like

$$\mathcal{E}\dagger\ldots\dagger,\kappa\dagger\ldots\dagger$$

selecting exactly one from however many, so they make a good representation of de Bruijn variables. Let us refer to such things by Roman letters (x, y).

Of course, as variables are thinnings, for any such $x \in \kappa \leftarrow \gamma$, we may construct $\widehat{x} \in \overline{x} \leq \gamma$ and observe that every variable in $\kappa' \leftarrow \gamma$ is either x (in which case $\kappa = \kappa'$) or some $y;\widehat{x}$, where $y \in \kappa' \leftarrow \overline{x}$. Let us consider ourselves entitled to treat x and $y;\widehat{x}$ as a valid covering of patterns for $\kappa' \leftarrow \gamma$.

A.2 Terms with Schematic Variables

The *terms* $\Sigma^{\mathsf{t}}(\gamma,\kappa)$ of kind κ in scope γ, and the *spines* $\Sigma^{\mathsf{s}}(\gamma,\delta)$ for scope δ in scope γ, are given thus:

$$\begin{array}{rcl} \Sigma^{\mathsf{t}}(\gamma,[\kappa]\kappa') & = & \Sigma^{\mathsf{t}}((\gamma,\kappa),\kappa') \\ \Sigma^{\mathsf{t}}(\gamma,\iota) & = & \Sigma^{\mathsf{c}}(\Sigma^{\mathsf{t}}(\gamma,-),\iota) \\ & \cup & \{x[ss] \mid x \in [\delta]\iota \leftarrow \gamma,\ ss \in \Sigma^{\mathsf{s}}(\gamma,\delta)\} \\ \\ \Sigma^{\mathsf{s}}(\gamma,\mathcal{E}) & = & \{\mathcal{E}\} \\ \Sigma^{\mathsf{s}}(\gamma,(\delta,\kappa)) & = & \{ss,t \mid ss \in \Sigma^{\mathsf{s}}(\gamma,\delta),\ t \in \Sigma^{\mathsf{t}}(\gamma,\kappa)\} \end{array}$$

When a variable has object sort, $x \in \iota \leftarrow \Gamma$, its argument spine is empty, and we may abbreviate the term $x[\mathcal{E}]$ to plain old x. The idea is that to construct a term of a given kind, we bring all of its bound variables into scope, then give either a construction allowed by the signature or a variable suitably instantiated.

This definition is designed to ensure that $\Sigma^{\mathsf{t}}(-,\kappa)$ and $\Sigma^{\mathsf{s}}(-,\delta)$ are functors from the category of scopes and thinnings to the category of sets and functions. As you can see, the scope γ occurs only to the right of $\leq$ in the definition, allowing thinning to act by composition. Meanwhile, the only time the scope changes is to bind a variable with $-,\kappa$ which acts functorially on thinnings. Let us write $t\theta$ and $ss\theta$ for the actions of thinning θ on term t and spine ss, respectively.

For human readability, let us continue to mark the growing of the scope in example terms by binding *names*, $[x]term$. When we use a variable t of kind $[\delta]\iota$, we must instantiate it as $t[ss]$ with

a spine ss for the scope δ given by its kind. We may write t for $t[]$ when t's kind makes it of object sort without further instantiation. The β-contraction scheme looks like this:

$$\alpha\ (\epsilon\ (\lambda\ x.\, t[x])\ (\Pi\ S\ x.\ T[x]))\ s\ \leadsto_\beta\ \epsilon\ t[\epsilon\ s\ S]\ T[\epsilon\ s\ S]$$

The point is that the *schematic* variables in the presentation of the contractions are now directly represented as variables in our syntax. Some of them have kinds which are not just object sorts, indicating that they bind variables and must be instantiated in the construction of object terms.

A.3 Parallel Action, Hereditary Substitution

We shall need to do more to terms than mere thinning. In particular, we shall need to instantiate variables with terms of appropriate kind. For example, when we instantiate β-contraction, we shall need to replace t with some actual term that binds a variable, putting $\epsilon\ s\ S$ in the place of that bound variable. Substitution begets substitution: will it ever end? We know from Watkins et al. [Watkins et al.(2003)Watkins, Cervesato, Pfenning, and Walker] that hereditary substitution for the logical framework amounts to $\beta\eta$-long-normalization for simply typed λ-calculus, and thus terminates satisfactorily. Andreas Abel [Abel(2006)] used Agda's *sized types* to implement *simultaneous* (replacing all variables in scope) hereditary substitution for simply typed λ-calculus. I edited the volume in which that paper appears, so it should not be a surprise that an anonymous reviewer offered an implementation of *single* (replacing just one variable in scope) hereditary substitution without sized types, using only *structural recursion* (on the type of the substituted variable). The grapevine being what it is, that program was the seed for C. Keller and Altenkirch's normalization formalization, with full soundness and completeness proofs [Keller and Altenkirch(2010)].

The usual approach is to make sure thinning works (which is clear from our construction) and then use thinning to manage the de Bruijn shifting required for substitution (when a term is used in a larger scope than the scope that it came from): it has long been observed [Goguen and Mckinna(1997), McBride(2000)] that thinning (or renaming, more generally) and substitution can be implemented by instantiating the parameters of *one* structural recursion on terms in *two* different ways (accounting for how to act on a variable and how to go under a binder).

Here, then, is how to implement *simultaneous* (no need to focus on one variable!) hereditary substitution (or rather, something slightly more general) for our simply kinded calculus by structural recursion on *scopes* (no sized types!) in one pass (no need for a prior implementation of thinning!).

The key is to define a notion of *parallel action*, explaining how to either thin or substitute each variable. Read $\gamma\,[\zeta\rangle\,\delta$ as an action mapping γ-terms to δ-terms by substituting some subscope $\zeta \leq \gamma$ and thinning the rest.

$$\frac{}{\mathcal{E} \in \mathcal{E}\,[\mathcal{E}\rangle\,\mathcal{E}} \qquad \frac{\sigma \in \gamma\,[\zeta\rangle\,\delta}{\sigma, \kappa \in (\gamma, \kappa)\,[\zeta\rangle\,(\delta, \kappa)} \qquad \frac{\sigma \in \gamma\,[\zeta\rangle\,\delta}{\sigma\dagger_\kappa \in \gamma\,[\zeta\rangle\,(\delta, \kappa)} \qquad \frac{\sigma \in \gamma\,[\zeta\rangle\,\delta \quad s \in \Sigma^{\mathsf{t}}(\delta, \kappa)}{\sigma/s \in (\gamma, \kappa)\,[\zeta, \kappa\rangle\,\delta}$$

The first three rules tell us that every thinning is an action; the fourth tells us that actions can also substitute. Crucially, given some $\theta \in \delta_0 \leq \delta$ and some $ss \in \Sigma^{\mathsf{s}}(\delta, \zeta)$, we may construct $\theta + ss \in \delta_0, \zeta\,[\zeta\rangle\,\delta$, just by copying θ to get an action in $\delta_0\,[\mathcal{E}\rangle\,\delta$, then adding the terms from the spine, extending the source scope to δ_0, ζ and the substitutive subscope to ζ, keeping the target at δ (which is the scope of the terms in the spine).

Now let us define

$$\frac{t \in \Sigma^{\mathsf{t}}(\gamma, \kappa) \quad \sigma \in \gamma\,[\zeta\rangle\,\delta}{t\{\kappa|\zeta\}\sigma \in \Sigma^{\mathsf{t}}(\delta, \kappa)} \qquad \frac{ss \in \Sigma^{\mathsf{s}}(\gamma, \delta') \quad \sigma \in \gamma\,[\zeta\rangle\,\delta}{ss\{\delta'|\zeta\}\sigma \in \Sigma^{\mathsf{s}}(\delta, \delta')}$$

$$\frac{x \in [\zeta_x]\iota \leftarrow \gamma \quad \sigma \in \gamma\,[\zeta\rangle\,\delta_0 \quad \theta \in \delta_0 \leq \delta \quad ss \in \Sigma^{\mathsf{s}}(\delta, \zeta_x)}{\mathsf{vact}\{\zeta\}\ x\ \sigma\ \theta\ ss \in \Sigma^{\mathsf{t}}(\delta, \iota)}$$

In casual usage, I shall omit the details given in $\{\ldots\}$, but for purposes of definition, it is crucial

to observe them, as they justify the recursion.

$$
\begin{array}{rcll}
t\{\zeta|[\kappa]\kappa'\}\sigma & = & t\{\zeta|\kappa'\}(\sigma,\kappa) & \\
c\,\vec{t}\ \ \{\zeta|\iota\}\ \ \sigma & = & c\,\vec{t\{\zeta|-\}\sigma} & \text{functoriality of } \Sigma^{\mathsf{c}}(-,\iota) \\
x[ss]\ \ \{\zeta|\iota\}\ \ \sigma & = & \mathsf{vact}\{\zeta\}\ x\ \sigma\ \delta\ (ss\{\zeta|\zeta_x\}\sigma) & \text{where } ss \in \Sigma^{\mathsf{s}}(\gamma,\zeta_x) \\
\\
\mathsf{vact}\{\zeta\}\ \ x\ \ (\sigma\dagger_\kappa)\ \theta\ ss & = & \mathsf{vact}\{\zeta\}\ x & \sigma\ (\dagger_\kappa;\theta)\ ss \\
\mathsf{vact}\{\zeta\}\ \ (x\dagger_\kappa)\ \ (\sigma\dagger,\kappa)\ \theta\ ss & = & \mathsf{vact}\{\zeta\}\ x\ \sigma\ (\dagger_\kappa;\theta)\ ss & \\
\mathsf{vact}\{\zeta,\kappa\}\ \ (x\dagger_\kappa)\ \ (\sigma/t)\ \theta\ ss & = & \mathsf{vact}\{\zeta\}\ x\ \sigma\ \theta\ ss & \\
\mathsf{vact}\{\zeta\}\ \ (x,\kappa)\ \ (\sigma\dagger,\kappa)\ \theta\ ss & = & (\widehat{(},\kappa);\theta)[ss] & \text{act by thinning} \\
\mathsf{vact}\{\zeta,[\zeta_x]\iota\}(x,[\zeta_x]\iota)(\sigma\dagger/t)\ \theta\ ss & = & t\{\zeta_x|\iota\}(\theta+ss) & \text{act by substituting} \\
\\
\mathcal{E}\ \ \{\zeta|\mathcal{E}\}\ \ \sigma & = & \mathcal{E} & \\
(ss,s)\,\{\zeta|\delta',\kappa\}\,\sigma & = & ss\{\zeta|\delta'\}\sigma, s\{\zeta|\kappa\}\sigma &
\end{array}
$$

The recursion is lexicographic, first on ζ, then on terms. Thinning an action to go under a binder does not alter ζ. As soon as we hit $x[ss]$, we substitute the spine immediately, then we search the action, σ to find the fate of x. The vact operation tears down σ and x, building up a thinning, θ. As soon as the variable we are looking for is found on top, we must either thin it and reattach the spine, or substitute it, acting hereditarily on its image with an action constructed from the spine. As vact searches, ζ is torn down, and in the hereditary substitution case, the substituted scope for the new substitution sits within the kind of the substituted variable.

References

[Abel(2006)] Andreas Abel. Implementing a normalizer using sized heterogeneous types. In Conor McBride and Tarmo Uustalu, editors, *Workshop on Mathematically Structured Functional Programming, MSFP@MPC 2006, Kuressaare, Estonia, July 2, 2006.*, Workshops in Computing. BCS, 2006. URL `http://ewic.bcs.org/content/ConWebDoc/5336`.

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

[Goguen and Mckinna(1997)] Healfdene Goguen and James Mckinna. Candidates for substitution, 1997.

[Keller and Altenkirch(2010)] Chantal Keller and Thorsten Altenkirch. Hereditary substitutions for simple types, formalized. In Venanzio Capretta and James Chapman, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010.*, pages 3–10. ACM, 2010. ISBN 978-1-4503-0255-5. doi: 10.1145/1863597.1863601. URL `http://doi.acm.org/10.1145/1863597.1863601`.

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

[McBride(2000)] Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, UK, 2000. URL `http://hdl.handle.net/1842/374`.

[Pierce and Turner(2000)] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[Watkins et al.(2003)Watkins, Cervesato, Pfenning, and Walker] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2003.