Outrageous but Meaningful Coincidences

Dependent type-safe syntax and evaluation

Conor McBride
University of Strathclyde
conor@cis.strath.ac.uk

Abstract

Tagless interpreters for well-typed terms in some object language are a standard example of the power and benefit of precise indexing in types, whether with dependent types, or generalized algebraic datatypes. The key is to reflect object language types as indices (however they may be constituted) for the term datatype in the host language, so that host type coincidence ensures object type coincidence. Whilst this technique is widespread for simply typed object languages, dependent types have proved a tougher nut with nontrivial computation in type equality. In their type-safe representations, Danielsson [2006] and Chapman [2009] succeed in capturing the equality rules, but at the cost of representing equality derivations explicitly within terms. This article delivers a type-safe representation for a dependently typed object language, dubbed KIPLING, whose computational type equality just appropriates that of its host, Agda. The KIPLING interpreter example is not merely *de rigeur*—it is key to the construction. At the heart of the technique is that key component of generic programming, the *universe*.

1. Introduction

Last century, we learned from Altenkirch and Reus [1999] how to represent simply typed terms precisely as an inductive family of datatypes [Dybjer 1991] in a dependently typed language. The idea is to make the type system of the host language police the typing rules of the object language by indexing the datatype representing object terms with a representation of object types. The payoff is that programs which manipulate the object language can take type safety for granted—well-typedness of object language terms becomes a matter of basic hygiene for the host. There is a rich literature of work which exploits this technique, both in the dependently typed setting and in Haskell-like languages with sufficiently precise typing mechanisms. For a small selection, see Baars and Swierstra [2004]; Brady and Hammond [2006]; Carette et al. [2009]; Chen and Xi [2003]; Pasalic et al. [2002].

This paper makes the jump to representing dependently typed object languages in a precise type-safe manner, a problem complicated by the fact that object language type equality requires nontrivial computation. However, the host language type system also boasts a computational equality: let us steal it. If we can push object language computation into host language types, object type equality becomes a matter of coincidence, specified by the outrageously simple method of writing the same variable in two places! The key is to choose our coincidences with care: we should not ask for coincidence in what types *say* if we only want coincidence in what types *mean*. Once we have found what host type captures the 'meaning' of object types, we can index by it and proceed as before.

The representation recipe. Recall the basic type-safe encoding method, working in Agda [Norell 2008]. First, define types.

```
data ⋆ : Set where
  ι   : ⋆
  _▷_ : ⋆ → ⋆ → ⋆
```

Next, define contexts, with de Bruijn [1972] indices typed as witnesses to context membership.

```
data Cx : Set where
  ε   : Cx
  _,_ : Cx → ⋆ → Cx
data _∋_ : Cx → ⋆ → Set where
  top : ∀ {Γ τ}     → Γ , τ ∋ τ
  pop : ∀ {Γ σ τ} → Γ ∋ τ → Γ , σ ∋ τ
```

Finally, define terms by giving an indexed syntax reflecting the typing rules which, fortunately, are syntax-directed. I make the traditional use of comment syntax to suggest typing rules.

```
data _⊢_ : Cx → ⋆ → Set where
  -- variables witness context membership
  var : ∀ {Γ τ}   → Γ ∋ τ
                  ---------
                  → Γ ⊢ τ
  -- λ-abstraction extends the context
  lam : ∀ {Γ σ τ} → Γ , σ ⊢ τ
                  -----------
                  → Γ ⊢ σ ▷ τ
  -- application demands a type coincidence
  app : ∀ {Γ σ τ} → Γ ⊢ σ ▷ τ → Γ ⊢ σ
                  -------------------
                  → Γ ⊢ τ
```

Notice how, for app, the domain of the function and the argument type must coincide. Agda will reject apps unless the candidates for σ are definitionally equal in type $\star$: it's really checking types. Moreover, with 'implicit syntax' [Norell 2007] combining insights from Damas and Milner [1982] via Pollack [1992] with *pattern unification* from Miller [1991], Agda makes a creditable effort at type *inference* for object language terms!

McKinna and I [McBride and McKinna 2004] showed how to write a typechecker which yields typed terms from raw preterms,

establishing coincidence by testing object-type equality. Augustsson and Carlsson [1999] showed how such guarantees can deliver a *tagless* interpreter, with no rechecking of types during execution.

Let us recall the construction. First, interpret the types, explaining what their *values* are. For simplicity, I interpret the base type as the natural numbers.

```
⟦_⟧⋆ : ⋆ → Set
⟦ι⟧⋆       = ℕ
⟦σ ▷ τ⟧⋆   = ⟦σ⟧⋆ → ⟦τ⟧⋆
```

Next, lift this interpretation to contexts, giving a suitable notion of *environment*—a tuple of values equipped with projection.

```
⟦_⟧ᶜ : Cx → Set
⟦ℰ⟧ᶜ     = 1
⟦Γ , σ⟧ᶜ = ⟦Γ⟧ᶜ × ⟦σ⟧⋆
⟦_⟧∋ : ∀ {Γ τ} → Γ ∋ τ → ⟦Γ⟧ᶜ → ⟦τ⟧⋆
⟦top⟧∋   (γ , t) = t
⟦pop i⟧∋ (γ , s) = ⟦i⟧∋ γ
```

Finally, interpret terms as functions from environments to values.

```
⟦_⟧⊢ : ∀ {Γ τ} → Γ ⊢ τ → ⟦Γ⟧ᶜ → ⟦τ⟧⋆
⟦var i⟧⊢   γ = ⟦i⟧∋ γ
⟦lam t⟧⊢   γ = λ s → ⟦t⟧⊢ (γ , s)
⟦app f a⟧⊢ γ = ⟦f⟧⊢ γ (⟦a⟧⊢ γ)
```

However, if we try to replay this game for *dependent* types, we get into a bit of a tangle. The recipe 'first types, next contexts, finally terms' seems to be thwarted by the infernal way that everything depends on everything else: contexts contain types, but later types must be validated with respect to earlier contexts; terms inhabit types, but dependent types mention terms. When all you have is a hammer, dependent type theory looks like a screw.

Brave souls undeterred press on. Boldly, we might attempt a *mutual* definition. Here is a fragment showing how type-level computation (here, If) really mixes everything up.

```
mutual
  data Cx : Set where
    ℰ   : Cx
    _,_ : (Γ : Cx) → Ty Γ → Cx
  data Ty : Cx → Set where
    Π   : ∀ {Γ} → (S : Ty Γ) → Ty (Γ , S)
                ---------------------
                → Ty Γ
    2   : ∀ {Γ} ---------
                → Ty Γ
    If  : ∀ {Γ} → Γ ⊢ 2 → Ty Γ → Ty Γ
                ---------------------
                → Ty Γ
    -- and more
  data _⊢_ : (Γ : Cx) → Ty Γ → Set where
    lam : ∀ {Γ S T} → Γ , S ⊢ T
                    ---------
                    → Γ ⊢ Π S T
    app : ∀ {Γ S T} → Γ ⊢ Π S T → (s : Γ ⊢ S)
                    ---------------------
                    → Γ ⊢ T [ s ]
    tt ff : ∀ {Γ}   ---------
                    → Γ ⊢ 2
    -- and more
```

Can you see trouble? Let me point out the main problems:

- *What are these peculiar mutual datatypes?* They are successively indexed by those which have gone before. This kind of definition, called "inductive-inductive" by analogy with "inductive-recursive", has no standard presentation in the literature, although Forsberg and Setzer [2010] have recently made valuable progress in this direction. Agda fails to forbid them, but that does not mean that they make sense.
- *Where have I hidden the variables?* We could try to proceed as before, but we immediately hit a snag.

```
data _∋_ : (Γ : Cx) → Ty Γ → Set where
  top : ∀ {Γ T}     → Γ , T ∋ T   -- bad
  pop : ∀ {Γ S T} → Γ ∋ T → Γ , S ∋ T   -- bad
```

  In both cases, we have T : Ty Γ in Γ , T or Γ ∋ T. However, in the return type, we need to put T in a *longer context*, but we find T ∌ Ty (Γ , −). We need some mechanism for *weakening*, but what? Can we define it as a function, extending our mutual definition in an even more peculiar way? We could make weakening explicit with a constructor

$$^{*} : \forall \{\Gamma\ S\} \to \mathsf{Ty}\ \Gamma \to \mathsf{Ty}\ (\Gamma , S)$$

  at the cost of extending the object language with an artefact of the encoding.
- *What is this T [s]?* In the application rule, we need to substitute the topmost bound variable in T with the term s. Again we must decide how to implement this, by defining a function or extending the syntax.
- *What coincidence does application demand?* Here, the domain of the function must equal the argument type *syntactically*. We might hope that $G \vdash$ If tt 2 2 and $G \vdash$ 2 would be compatible, but sadly If tt 2 2 and 2 are differently constructed wooden lumps of syntax, unanimated by computation. Of course, adding explicit syntax for weakening and substitution provides yet more ways for the same type to look different. To be suitably liberal, we must also define type equality Γ ⊢ S ≡ T (necessitating also value equality), then add something like the 'conversion rule'.

```
cast : ∀ {Γ S T} → Γ ⊢ S → Γ ⊢ S ≡ T
                   ---------------
                   → Γ ⊢ T
```

  We now have an object language with explicit casts and equality evidence, where these are silent in the host language.

These are hard problems, and I applaud both Danielsson and Chapman for taking them on. Their work—in particular, Chapman's thesis [Chapman 2008]—has much to teach us about the principles and the pragmatics of deep embeddings for dependently typed terms. But I, being a shallow sort of fellow, propose instead to cheat. In particular, rather than developing my own theory of equality, I plan to steal one. As a result, I shall be able to give a dependently typed object language whose encoding fits its syntax, with no artefacts for weakening or substitution, and no equality evidence. Moreover, I shall use nothing more remarkable than indexed inductive-recursive definitions—these are quite remarkable, actually, but they do have a set-theoretic model [Dybjer and Setzer 2001]. The resulting system is not, I must confess, entirely satisfactory, but it is a reassuringly cheap step in a useful direction.

2. What is KIPLING?

'Do you like Kipling?
'I don't know, you naughty boy, I've never kippled!'
from a postcard by Donald McGill

$$\frac{}{\mathcal{E} \vdash \text{VALID}} \qquad \frac{\Gamma \vdash \text{VALID} \quad \Gamma \vdash S\ \text{TYPE}}{\Gamma, x{:}S \vdash \text{VALID}}\ x \notin \Gamma$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \{\text{ZERO}, \text{ONE}, \text{TWO}, \text{NAT}\}\ \text{TYPE}} \qquad \frac{\Gamma \vdash S\ \text{TYPE} \quad \Gamma, x{:}S \vdash T\ \text{TYPE}}{\Gamma \vdash \{\text{SG}, \text{PI}\}\ x{:}S\ .\ T\ \text{TYPE}} \qquad \frac{\Gamma \vdash b : 2 \quad \Gamma \vdash \{T, F\}\ \text{TYPE}}{\Gamma \vdash \text{IF}\ b\ T\ F\ \text{TYPE}}$$

$$\frac{\Gamma \vdash b : 2 \quad \Gamma \vdash \{T, F\}\ \text{TYPE}}{\Gamma \vdash \text{IF TT}\ T\ F \equiv T} \qquad \frac{\Gamma \vdash b : 2 \quad \Gamma \vdash \{T, F\}\ \text{TYPE}}{\Gamma \vdash \text{IF FF}\ T\ F \equiv F} \qquad \frac{\Gamma \vdash b \equiv b' : 2 \quad \Gamma \vdash T \equiv T' \quad \Gamma \vdash F \equiv F'}{\Gamma \vdash \text{IF}\ b\ T\ F \equiv \text{IF}\ b'\ T'\ F'}$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \text{VOID} : \text{ONE}} \qquad \frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \{\text{TT}, \text{FF}\} : \text{TWO}} \qquad \frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \text{ZE} : \text{NAT}} \qquad \frac{\Gamma \vdash n : \text{NAT}}{\Gamma \vdash \text{SU}\ n : \text{NAT}}$$

$$\frac{\Gamma, x{:}S \vdash T\ \text{TYPE} \quad \Gamma \vdash s : S \quad \Gamma \vdash t : T[s]}{\Gamma \vdash s\ \&\ t : \text{SG}\ x{:}S\ .\ T} \qquad \frac{\Gamma, x{:}S \vdash t : T}{\Gamma \vdash \text{LAM}\ x\ .\ t : \text{PI}\ x{:}S\ .\ T} \qquad \frac{\Gamma \vdash s : S \quad \Gamma \vdash S \equiv T}{\Gamma \vdash s : T}$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash x : S}\ x{:}S \in \Gamma$$

$$\frac{\Gamma \vdash z : \text{ZERO} \quad \Gamma \vdash \Gamma \vdash T\ \text{TYPE}}{\Gamma \vdash \text{MAGIC}\ z\ T : T}$$

$$\frac{\Gamma \vdash b : \text{TWO} \quad \Gamma, x{:}\text{TWO} \vdash P\ \text{TYPE} \quad \Gamma \vdash t : P[\text{TT}] \quad \Gamma \vdash f : P[\text{FF}]}{\Gamma \vdash \text{IF}\ b\ x.P\ t\ f : P[b]}$$

$$\frac{\Gamma \vdash n : \text{NAT} \quad \Gamma, x{:}\text{NAT} \vdash P\ \text{TYPE} \quad \Gamma \vdash z : P[\text{ZE}] \quad \Gamma \vdash s : \text{PI}\ n{:}\text{NAT}\ .\ \text{PI}\ h{:}P[n]\ .\ P[\text{SU}\ n]}{\Gamma \vdash \text{REC}\ n\ x.P\ z\ s : P[n]}$$

$$\frac{\Gamma \vdash p : \text{SG}\ x{:}S\ .\ T}{\Gamma \vdash \text{FST}\ p : S} \qquad \frac{\Gamma \vdash p : \text{SG}\ x{:}S\ .\ T}{\Gamma \vdash \text{SND}\ p : T[\text{FST}\ p]}$$

$$\frac{\Gamma \vdash f : \text{PI}\ x{:}S\ .\ T \quad \Gamma \vdash s : S}{\Gamma \vdash f\ \$\ s : T[s]}$$

$$\frac{\Gamma, x{:}\text{TWO} \vdash P\ \text{TYPE} \quad \Gamma \vdash t : P[\text{TT}] \quad \Gamma \vdash f : P[\text{FF}]}{\Gamma \vdash \text{IF TT}\ x.P\ t\ f \equiv t : P[\text{TT}] \quad \Gamma \vdash \text{IF FF}\ x.P\ t\ f \equiv f : P[\text{FF}]}$$

$$\frac{\Gamma \vdash n : \text{NAT} \quad \Gamma, x{:}\text{NAT} \vdash P\ \text{TYPE} \quad \Gamma \vdash z : P[\text{ZE}] \quad \Gamma \vdash s : \text{PI}\ n{:}\text{NAT}\ .\ \text{PI}\ h{:}P[n]\ .\ P[\text{SU}\ n]}{\Gamma \vdash \text{REC ZE}\ x.P\ z\ s \equiv z : P[\text{ZE}] \quad \Gamma \vdash \text{REC}\ (\text{SU}\ n)\ x.P\ z\ s \equiv s\ \$\ n\ \$\ \text{REC}\ n\ x.P\ z\ s : P[\text{SU}\ n]}$$

$$\frac{\Gamma, x{:}S \vdash T\ \text{TYPE} \quad \Gamma \vdash s : S \quad \Gamma \vdash t : T[s]}{\Gamma \vdash \text{FST}\ (s\ \&\ t) \equiv s : S \quad \Gamma \vdash \text{SND}\ (s\ \&\ t) \equiv t : T[s]}$$

$$\frac{\Gamma \vdash \Gamma, x{:}S \vdash t : T \quad \Gamma \vdash s : S}{\Gamma \vdash (\text{LAM}\ x\ .\ t)\ \$\ s \equiv t[s] : T[s]}$$

Figure 1. The KIPLING rules (omitting some structural and equivalence closure rules in type and value equality)

Let us fix a dependently typed language to model. KIPLING is a rudimentary dependent type theory with the minimal non-trivial feature for computing types—IF.[1] The (scope-annotated) syntax of KIPLING types and terms is as follows. I have left-justified the *canonical* (introduction) forms and right-justified the *non-canonical* (elimination) forms.

$$\begin{array}{rcl}
\langle var_{\Gamma,x} \rangle & ::= & x \\
& | & \langle var_{\Gamma} \rangle \\
\langle ty_{\Gamma} \rangle & ::= & \text{ZERO} \mid \text{ONE} \mid \text{TWO} \mid \text{NAT} \\
& | & \text{SG}\ x : \langle ty_{\Gamma} \rangle\ .\ \langle ty_{\Gamma,x} \rangle \\
& | & \text{PI}\ x : \langle ty_{\Gamma} \rangle\ .\ \langle ty_{\Gamma,x} \rangle \\
& | & \text{IF}\ \langle tm_{\Gamma} \rangle\ \langle ty_{\Gamma} \rangle\ \langle ty_{\Gamma} \rangle \\
\langle tm_{\Gamma} \rangle & ::= & \langle var_{\Gamma} \rangle \\
& | & \text{MAGIC}\ \langle tm_{\Gamma} \rangle\ \langle ty_{\Gamma} \rangle \\
& | & \text{VOID} \\
& | & \text{TT} \mid \text{FF} \mid \text{IF}\ \langle tm_{\Gamma} \rangle\ x.\langle ty_{\Gamma,x} \rangle\ \langle tm_{\Gamma} \rangle\ \langle tm_{\Gamma} \rangle \\
& | & \text{ZE} \mid \text{SU}\ \langle tm_{\Gamma} \rangle \\
& | & \text{REC}\ \langle tm_{\Gamma} \rangle\ x.\langle ty_{\Gamma,x} \rangle\ \langle tm_{\Gamma} \rangle\ \langle tm_{\Gamma} \rangle \\
& | & \langle tm_{\Gamma} \rangle\ \&\ \langle tm_{\Gamma} \rangle \mid \text{FST}\ \langle tm_{\Gamma} \rangle \mid \text{SND}\ \langle tm_{\Gamma} \rangle \\
& | & \text{LAM}\ x.\langle tm_{\Gamma} \rangle \mid \langle tm_{\Gamma} \rangle\ \$\ \langle tm_{\Gamma} \rangle
\end{array}$$

Figure 1 shows the typing rules for KIPLING. The equality rules for IF make type equality computational, and IF's structural rule makes type equality depend on value equality. I use a set notation to denote multiple premises and conclusions, and omit structural and equivalence closure rules, for brevity.

If you are keen to try KIPLING, I give a shallow embedding into Agda in Figure 2, declaring the canonical components and defining the rest. I omit a definition for PI, because Agda already functions. Note that I use the *large* Agda function space wherever KIPLING binds a variable to give a dependent type.

[1] Rudyard Kipling's poem 'If...' has inspired many youths to greatness.

```
data Zero : Set where
magic : Zero → (X : Set) → X
magic ()

record 1 : Set where
  constructor void

data 2 : Set where
  tt : 2
  ff : 2

If : 2 → Set → Set → Set
If tt T F = T
If ff T F = F

if : (b : 2) → (P : 2 → Set) → P tt → P ff → P b
if tt P t f = t
if ff P t f = f

data ℕ : Set where
  ze : ℕ
  su : ℕ → ℕ

rec : (n : ℕ) → (P : ℕ → Set) →
      P ze → ((n : ℕ) → P n → P (su n)) → P n
rec ze     P z s = z
rec (su n) P z s = s n (rec n P z s)

record Σ (S : Set) (T : S → Set) : Set where
  constructor _,_
  field
    fst : S
    snd : T fst
```

Figure 2. A shallow embedding of KIPLING in Agda

3. The KIPLING Universe

We may not know how to represent a dependently typed syntax, but we can at least collect a *universe*—in the sense of Martin-Löf [1984]—of the canonical KIPLING types, by means of an *inductive-recursive* definition. That is, we define simultaneously a datatype U which encodes the sets and a function **El** which decodes the codes. This is an entirely standard technique, indeed, a key motivating example for induction-recursion [Dybjer and Setzer 1999].

```
mutual
  data U : Set where
    'Zero' '1' '2' 'ℕ' : U
    'Π' 'Σ' : (S : U) → (El S → U) → U
  El : U → Set
  El 'Zero'      = Zero
  El '1'         = 1
  El '2'         = 2
  El 'ℕ'         = ℕ
  El ('Π' S T)   = (s : El S) → El (T s)
  El ('Σ' S T)   = Σ (El S) λ s → El (T s)
```

Observe the crucial use of **El** to encode value dependency functionally in the definition of U. Please note, this use of **El** $S \to$ U is not 'higher-order abstract syntax' in the sense of the Logical Framework [Harper et al. 1993]. Here, $\to$ denotes the full Agda function space, giving us a value we can compute with, not just a variable we can place. We can certainly define the non-dependent variants of 'Π' and 'Σ'.

```
_'→'_ : U → U → U
S '→' T = 'Π' S λ _ → T
_'×'_ : U → U → U
S '×' T = 'Σ' S λ _ → T
```

However, the power of the function space also allows us to equip U with IF, even though it is not a constructor—and all the better so, for it computes!

```
'If' : 2 → U → U → U
'If' tt T F = T
'If' ff T F = F
```

What we have done here is to fix in U the meanings of the *closed* KIPLING types. We have not yet given a syntax for types which gives *representation* to things in U, but we have determined a target for interpreting that syntax. Be clear that, thanks to the full power of $\to$, U contains many types which are not expressible as KIPLING types. For example, we could define

```
'Vec' : U → ℕ → U
'Vec' X ze     = '1'
'Vec' X (su n) = X '×' 'Vec' X n
```

and then form

```
'Π' 'ℕ' λ n → 'Vec' '2' n : U
```

but KIPLING does not provide a type-level recursor.

To give a syntax, we must somehow name the KIPLING-presentable creatures from this jungle. Before we can come to terms with syntax, however, we must develop an understanding of variables, and the contexts which account for them.

4. Dependent Contexts and Typed Variables

Contexts assign types to free variables. Each context determines a set of *environments*—tuples of values which the corresponding variables may simultaneously take. Contexts in dependent type systems form what de Bruijn [1991] calls *telescopes*, where each successive type may mention—i.e. depend on the *values* of—all previous variables. We may thus specify such a dependent set as a function from an environment to U. Correspondingly, dependent contexts naturally lend themselves to inductive-recursive definition.

```
mutual
  data Cx : Set where
    '𝓔' : Cx
    _,_ : (Γ : Cx) → (⟦Γ⟧ᶜ → U) → Cx
  ⟦_⟧ᶜ : Cx → Set
  ⟦'𝓔'⟧ᶜ     = 1
  ⟦Γ , S⟧ᶜ  = Σ ⟦Γ⟧ᶜ λ γ → El (S γ)
```

The semantic object corresponding to a 'Γ-type' is a function in $\llbracket\, \Gamma \,\rrbracket^{\mathsf{C}} \to$ U, giving a closed type for any given environment. This functional presentation will act as the carrier for a more precise shallow embedding of KIPLING types into our U, indirected via **El** into Agda's Set. Note that context extension Γ, S ensures that S is indeed a Γ-type.

Digression—left- versus right-nesting. The above notion of context is not news to anyone who has seen Pollack's treatment of dependent record types. Pollack [2002] notes that these left-nested records have a fixed field structure but dependent field types, where their right-nested counterparts permit variant structures.

```
data Rec : Set₁ where
  ℰ   : Rec
  _,_ : (S : Set) → (S → Rec) → Rec
⟦_⟧ᴿ : Rec → Set
⟦ℰ⟧ᴿ     = 1
⟦S , R⟧ᴿ = Σ S λ s → ⟦R s⟧ᴿ
```

This Rec does not rely on induction recursion, but it is necessarily *large*, as it packs up Sets themselves, rather than their codes. Again, because the $S \to$ Rec is full functional abstraction, not just variable-binding, the value of each field determines the whole field structure of what follows, not just the types of the fields. This power to code variant structures makes record right-nesting ideal for encoding nodes in datatypes, as Dybjer and Setzer [1999] do and we follow [Chapman et al. 2010], but rotten for encoding contexts. For one thing, we typically require ready access to the 'local' right end of a context. More seriously, you cannot define a type of references uniformly for a given R : Rec, only a notion of projection specific to each value in $\llbracket\, R \,\rrbracket^{\mathsf{R}}$. We rather expect a context to determine our choice of variables, good for any environment or none. ***End of digression.***

Let us indeed check that left-nested contexts give rise to a notion of variable. As we know what a Γ-type is, we can even repeat the Altenkirch and Reus [1999] method of indexing variables in Γ with the Γ-type they deliver.

```
data _∋_ : (Γ : Cx) (T : ⟦Γ⟧ᶜ → U) → Set where
  top : ∀ {Γ T}     → Γ , T ∋ T · fst
  pop : ∀ {Γ S T} → Γ ∋ T → Γ , S ∋ T · fst
```

What a blessed relief it is to index by the functional presentation of Γ-types! When we tried to index variables by the syntax of types, we had to give syntactic account of weakening, but here, $\cdot$ **fst** turns a function from a short environment to a function from an extended environment. We have recovered a suitably typed notion of de Bruijn index, ready for deployment as the variables of a type-safe syntax. Let us make sure that we can interpret our variables as projections from environments.

```
⟦_⟧∋ : ∀ {Γ T} → Γ∋T → (γ : ⟦Γ⟧ᶜ) → El (T γ)
⟦top⟧∋   (γ , t) = t
⟦pop i⟧∋ (γ , s) = ⟦i⟧∋ γ
```

Plus ça change, plus c'est la même chose! It is indeed a pleasure to see the very same function accepted at the more precise type, its journey leftward tallying precisely with the chain of **fst** projections at the type level.

5. A Combinatory Excursion

We are likely to spend most of the rest of this paper programming with functions from environments which take the form of tuples. It is worth making a small investment in combinators to tidy up our higher-order programming, keeping the plumbing under the floorboards wherever we can. Indeed I started a moment ago, writing the composition $T \cdot \mathbf{fst}$ for $\lambda\, \gamma \to T\ (\mathbf{fst}\ \gamma)$. Composition is familiar to functional programmers: in Haskell, we have

```
(.) :: (s -> t) -> (r -> s) -> (r -> t)
f . g = \ r -> f (g r)
```

But what is the type of composition for dependently typed functions? Martin-Löf challenged Norell at his thesis defence to give the following type:

```
_·_ : ∀ {a b c} {R : Set a} {S : R → Set b}
                {T : (r : R) → S r → Set c}
    → (∀ {r} (s : S r) → T r s)
    → (g : (r : R) → S r)
    → (r : R) → T r (g r)
f · g = λ r → f (g r)
```

As you can see, every opportunity for dependency is taken. The parameters R, S, T define a telescope. The polymorphism of f is exactly enough to cope with the dependent type of g. The definition also incorporates *universe polymorphism*, as witnessed by the Set levels a, b and c, so we can compose at any level.

Digression—type inference in Agda. How on earth can type inference cope with this complexity? It is not as tricky as one might at first think, because the variables abstracted by the unknowns are used at full generality in the types of f and g. To unify $(r : R) \to S\ r$ with, say, $(X : \mathsf{Set}) \to X \to \mathsf{Set}$, we must solve

$$\begin{array}{rcll} R & \equiv & \mathsf{Set} & \\ S\ r & \equiv & r \to \mathsf{Set} & \text{for all } r \end{array}$$

so a promising candidate solution is

$$\begin{array}{rcl} R & \mapsto & \mathsf{Set} \\ S & \mapsto & \lambda\, r \to r \to \mathsf{Set} \end{array}$$

Agda does attempt general higher-order unification: as Huet [1975] showed, it is undecidable, and worse, it does not yield unique solutions, hence it is hardly the thing to flesh out the clear intentions of the programmer. However, Miller [1991] observed that the special case where constraints on functions apply them to distinct universally quantified variables $f\ \vec{x} \equiv t$ yield candidate most general[2] solutions $f \mapsto \lambda \vec{x} \to t$, which succeed subject to scope conditions. Agda's type inference is based on Miller's unification, and can thus be expected to solve functional variables from general constraints. ***End of digression.***

The plumbing of an environment through a computation is something which the λ-calculus makes ugly, but combinatory logic takes in its stride [Curry and Feys 1958]. What do K and S do, if not relativise constants and application to an environment? Let us facilitate an applicative programming style relative to an environment [McBride and Paterson 2008], by equipping the traditional combinators for dependent types. The `ᴷ` combinator has just the type you might expect in the Hindley-Milner world.

```
ᴷ : ∀ {a b} {Γ : Set a} {X : Set b} → X → Γ → X
ᴷ = λ x γ → x
```

There is no way to make the type of the constant depend on an environment it discards!

However, the `_ˢ_` combinator more than compensates, again taking a full telescope of parameters. I make it a left-associative infix operator, as befits its usage for lifting application. I make these combinators very small, so that it is easy to ignore them and just gain the applicative intuition for what is going on.

```
_ˢ_ : ∀ {a b c}
        {Γ : Set a}
        {S : Γ → Set b}
        {T : (γ : Γ) → S γ → Set c}
    → (f : (γ : Γ) (s : S γ) → T γ s)
    → (s : (γ : Γ) → S γ)
    → (γ : Γ) → T γ (s γ)
f ˢ s = λ γ → f γ (s γ)
```

The type reflects the possibility that we are lifting a dependent function, whose Π-type itself depends on the environment.

We shall need two more paddles before we set forth for the creek. As the context grows, so the environment becomes a bigger tuple. We must write functions from environments, but we may prefer to do so in a curried style, for reasons both aesthetic and technical—the latter I shall attend to later. Hence we may need to apply dependent uncurrying (also known as the eliminator for Σ-types) to our functions.

```
ᵛ : ∀ {S T} {P : Σ S T → Set}
  → ((s : S) (t : T s) → P (s , t))
  → (st : Σ S T) → P st
ᵛ p (s , t) = p s t
```

The type shows the inevitable dependency at every opportunity, but the function is as we might expect.[3]

The inverse, dependent currying, will also prove useful.

```
^ : ∀ {a} {S T} {P : Σ S T → Set a}
  → ((st : Σ S T) → P st)
  → (s : S) (t : T s) → P (s , t)
^ p s t = p (s , t)
```

The names I have chosen for these operations show 'two become one' and 'one become two', respectively. Moreover, `^` gives the implementation of the lambda!

Note that

$$\texttt{\^{}}\, f\ \texttt{ˢ}\ s \equiv \lambda\, \gamma \to f\ (\gamma , s\ \gamma)$$

exactly corresponding to the instantiation of a bound variable.

Now, with heart and nerve and sinew, we are ready to carry on KIPLING!

6. How to Say what U you Mean

Recall that we recovered a workable presentation of de Bruijn variables by indexing with what types *mean*, not what they *say*. Let us apply the same approach to the syntax of types and terms. If we have a shallow embedding of KIPLING types as functions to U, we can use it to index a deep embedding of KIPLING types, effectively

[2] assuming the η law holds

[3] At present the 'lazy' variant `ᵛ p st = p (fst st) (snd st)` is the one that works. Agda erroneously makes the above *strict*, losing type preservation!

explaining which meanings can be said—a 'type-is-representable' predicate in the style of Crary et al. [1998]. We can also use *shallow* types to index a deep embedding of KIPLING *terms*, guaranteeing compatibility of the represented values. The catch is that we shall need to say what those values are, if we are to feed them to the functions which represent dependent types. A tagless interpreter is not just a nifty example, it is an essential component of the definition!

The deep embedding of KIPLING takes the form of an indexed inductive-recursive definition, with this signature:

```
mutual
  data _⋆_ (Γ : Cx) : (⟦Γ⟧ᶜ → U) → Set where
    -- Γ ⋆ T contains syntax representing T itself
  data _⊢_ (Γ : Cx) : (⟦Γ⟧ᶜ → U) → Set where
    -- Γ ⊢ T contains syntax representing T's inhabitants
  ⟦_⟧⊢ : ∀ {Γ T} → Γ ⊢ T → (γ : ⟦Γ⟧ᶜ) → El (T γ)
    -- tagless interpreter yields function from representation
```

For convenience, I define the datatypes of sets and terms mutually, but the old trick (in my thesis [McBride 1999] if nowhere else) of coding mutual definitions as single definitions indexed by choice of branch recovers indexed induction recursion in its standard form.

Representing the canonical types. Let us now explain which of our shallow functional types have a syntactic representation. We start with the primitive constants:

```
ZERO : Γ ⋆ᴷ 'Zero'
ONE  : Γ ⋆ᴷ '1'
TWO  : Γ ⋆ᴷ '2'
NAT  : Γ ⋆ᴷ 'ℕ'
```

Next, we add Π-types.

$$\frac{\text{PI} : \forall\,\{S\ T\} \to \Gamma \star S \to \Gamma\text{, } S \star T}{\to \Gamma \star^{\kappa} \text{'Π'}\ ^{\text{s}}\ S\ ^{\text{s}}{}^{\lambda}\ T}$$

Pleasingly, Agda can infer the types of S and the uncurried T. For the curious amongst you, they are as follows

$$S : \llbracket \Gamma \rrbracket^{\mathsf{C}} \to \mathsf{U}$$
$$T : \llbracket \Gamma, S \rrbracket^{\mathsf{C}} \to \mathsf{U} \quad \equiv \quad (\Sigma\ \llbracket \Gamma \rrbracket^{\mathsf{C}}\ \lambda\,\gamma \to \mathbf{El}\ (S\ \gamma)) \to \mathsf{U}$$

with T taking not only the environment, but also an S-value. Currying, we get

$$^{\lambda}\,T\ :\ (\gamma\ :\ \llbracket \Gamma \rrbracket^{\mathsf{C}}) \to \mathbf{El}\ (S\ \gamma) \to \mathsf{U}$$

so, plumbing the environment,

$$^{\kappa}\text{'Π'}\ ^{\text{s}}\ S\ ^{\text{s}}{}^{\lambda}\ T \quad \equiv \quad \lambda\,\gamma \to \text{'Π'}\ (S\ \gamma)\ \lambda\,s \to T\ (\gamma, s)$$

which is indeed a meaningful Γ-type. I have, I must confess, been slightly crafty: by choosing an uncurried T, I have arranged for the value of T to be readily inferrable, provided Agda can synthesize a type for the range. Had I chosen the curried variant, uncurrying T in the premise, thus

$$\frac{\text{PI} : \forall\,\{S\ T\} \to \Gamma \star S \to \Gamma\text{, } S \star^{\text{v}} T}{\to \Gamma \star^{\kappa} \text{'Π'}\ ^{\text{s}}\ S\ ^{\text{s}}\ T}$$

the type inference problem for a range with synthesized type $\Gamma, S \star T'$ would be

$$^{\text{v}}\,T \ \equiv\ T' \quad \text{unpacking to} \quad T\ (\mathbf{fst}\ \gamma)\ (\mathbf{snd}\ \gamma) \ \equiv\ T'\ \gamma$$

which is not solved by pattern unification. We expect the representation to determine the represented type, so type information will propagate outwards if we set things up accordingly.

The treatment of Σ-types follows the same plan.

$$\frac{\text{SG} : \forall\,\{S\ T\} \to \Gamma \star S \to \Gamma\text{, } S \star T}{\to \Gamma \star^{\kappa} \text{'Σ'}\ ^{\text{s}}\ S\ ^{\text{s}}{}^{\lambda}\ T}$$

We have syntax for our canonical types, reflecting the constructors of U directly, but what about KIPLING's IF? We have defined its meaning, **If**, by computation over U. Let us just provide the means to say it!

$$\frac{\text{IF} : \forall\,\{T\ F\} \to (b\ :\ \Gamma \vdash^{\kappa} \text{'2'}) \to \Gamma \star T \to \Gamma \star F}{\to \Gamma \star^{\kappa} \mathbf{'If'}\ ^{\text{s}}\ \llbracket b \rrbracket^{\vdash}\ ^{\text{s}}\ T\ ^{\text{s}}\ F}$$

This, our only form of dependency, we give a term to branch upon. To say which type we mean, we must interpret that term!

Digression—recursive large elimination. We cannot quite play the same game to compute types by recursion on ℕ. We may certainly implement a primitive recursor for U in Agda:

```
RecU : U → (ℕ → U → U) → ℕ → U
RecU Z S ze     = Z
RecU Z S (su n) = S b (RecU Z S n)
```

The trouble comes when we try to represent it. For IF, we could ask for representatives of T and F, but here we cannot ask for a representative of S. To do so, we would need S to be a function from environments to U, so we need to encode its argument types in a context: while we can encode ℕ as function to U, we cannot yet encode U itself! ***End of digression.***

Digression—the power of IF. Do not underestimate the power of dependent types with IF as the only mechanism for inspecting values. We may write Gödel's 'truth-predicate' IF b ONE ZERO, encoding any decidable property as a type of triumph or disaster. For undecidable properties with checkable *certificates*, Gödel-code the certificates, then write a certificate-checker as a function inhabiting PI NAT TWO and reflect the result with the truth-predicate. We may, of course, use SG to present propositions as the existence of certificates which check triumphantly. With IF, as Kipling put it, *'Yours is the Earth and everything that's in it'*; but it takes more to gain the universe, and *vice versa*. ***End of digression.***

Well typed terms with a tagless interpreter. Let us now give the terms, together with their interpretation as functions from environments to elements of the relevant type. Agda requires that we write the syntax and its interpreter in separate chunks of the mutual definition, but it helps to keep things with their meanings on paper. For the most part, we shall benefit from adopting a point-free style, quietly plumbing the environment with combinators.

Our typed de Bruijn indices give us variables, interpreted as projections.

$$\frac{\text{VAR} : \forall\,\{T\} \to \Gamma \ni T}{\to \Gamma \vdash T}$$
$$\llbracket\ \text{VAR}\ x\ \rrbracket^{\vdash} \ =\ \llbracket x \rrbracket^{\ni}$$

We have constructors for our (nonempty) base types, readily interpreted in the host language.

```
VOID  : Γ ⊢ᴷ '1'
TT FF : Γ ⊢ᴷ '2'
ZE    : Γ ⊢ᴷ 'ℕ'
SU    : Γ ⊢ᴷ 'ℕ' → Γ ⊢ᴷ 'ℕ'
⟦VOID⟧⊢ = ᴷvoid
⟦TT⟧⊢   = ᴷtt
⟦FF⟧⊢   = ᴷff
⟦ZE⟧⊢   = ᴷze
⟦SU n⟧⊢ = ᴷsu ˢ ⟦n⟧⊢
```

We can equip ZERO with an eliminator, giving us anything.

```
MAGIC : ∀ { T } → Γ ⊢ κ 'Zero' → Γ ⋆ T
                ———————————————————
                → Γ ⊢ T
```

Its interpretation dismisses the empty type.

```
⟦ MAGIC { T } z _ ⟧⊢ = κ magic ˢ ⟦ z ⟧⊢ ˢ (κ El ˢ T)
```

Meanwhile, ONE needs no eliminator, for it tells us nothing of interest.

The dependent eliminator for TWO delivers a type parametrized by a value. Although we have no object-level type of functions to U, we can achieve the same effect by *binding a variable*, specifying the result as a type in an extended context.

```
IF : ∀ { P } → (b : Γ ⊢ κ '2')
             → Γ , κ '2' ⋆ P
             → Γ ⊢ ∧ P ˢ κ tt → Γ ⊢ ∧ P ˢ κ ff
             ———————————————————————————————
             → Γ ⊢ ∧ P ˢ ⟦ b ⟧⊢
```

The P which gives the meaning to the return type is thus a function from a tuple—to use it with the implicit Γ-environment, we need merely curry it! This IF does indeed deliver P instantiated to any Boolean, provided it can be delivered for TT and FF. We may implement the function compactly with a helper function.

```
⟦ IF { P } b _ t f ⟧⊢ = ifHelp ˢ ⟦ b ⟧⊢ where
  ifHelp : (γ : _) → (b : 2) → El (P (γ , b))
  ifHelp γ tt = ⟦ t ⟧⊢ γ
  ifHelp γ ff = ⟦ f ⟧⊢ γ
```

We can give the induction principle for $\mathbb{N}$ in the same way. The 'induction predicate' P is a function over environments extended with a number—the combinatory shorthand struggles to cope with the extra variable binding in the step case, so I give the expansion.

```
REC : ∀ { P }
  → (n : Γ ⊢ κ 'ℕ')
  → Γ , κ 'ℕ' ⋆ P
  → Γ ⊢ ∧ P ˢ κ ze
  → Γ ⊢ (λ γ → 'Π' 'ℕ' λ n → P (γ , n) '→' P (γ , su n))
  ———————————————————————————————————————————————
  → Γ ⊢ ∧ P ˢ ⟦ n ⟧⊢

⟦ REC { P } n _ z s ⟧⊢ = recHelp ˢ ⟦ n ⟧⊢ where
  recHelp : (γ : _) → (n : ℕ) → El (P (γ , n))
  recHelp γ ze     = ⟦ z ⟧⊢ γ
  recHelp γ (su n) = ⟦ s ⟧⊢ γ n (recHelp γ n)
```

Let us turn now to the compound types. Here is pairing, which necessarily evaluates its first component to type its second.

```
_&_ : ∀ { S T } → (s : Γ ⊢ S) → Γ ⊢ T ˢ ⟦ s ⟧⊢
                ———————————————————————————
                → Γ ⊢ κ 'Σ' ˢ S ˢ T
```

We also need the projections.

```
FST : ∀ { S T } → Γ ⊢ κ 'Σ' ˢ S ˢ T
                ———————————————
                → Γ ⊢ S
SND : ∀ { S T } → (p : Γ ⊢ κ 'Σ' ˢ S ˢ T)
                ———————————————————————
                → Γ ⊢ T ˢ (λ γ → fst (⟦ p ⟧⊢ γ))
```

The type of the second projection depends on the value of the first, so I had hoped to write $\Gamma \vdash T\ ^{\mathsf{s}}\ (^{\kappa}\,\mathbf{fst}\ ^{\mathsf{s}}\ [\![\ p\ ]\!]^{\vdash})$, but inference failed, hence my explicit expansion. Moreover, the implementation requires expansion in all three cases.[4]

```
⟦ s & t ⟧⊢  = λ γ → ⟦ s ⟧⊢ γ , ⟦ t ⟧⊢ γ
⟦ FST p ⟧⊢ = λ γ → fst (⟦ p ⟧⊢ γ)
⟦ SND p ⟧⊢ = λ γ → snd (⟦ p ⟧⊢ γ)
```

Functional abstraction is straightforward, and application just follows the rule. Interpretation is bracket abstraction!

```
LAM : ∀ { S T } → Γ , S ⊢ ν T
                ———————————————
                → Γ ⊢ κ 'Π' ˢ S ˢ T
_$_ : ∀ { S T } → Γ ⊢ κ 'Π' ˢ S ˢ T → (s : Γ ⊢ S)
                ———————————————————————————————
                → Γ ⊢ T ˢ ⟦ s ⟧⊢
⟦ LAM t ⟧⊢ = ∧ ⟦ t ⟧⊢
⟦ f $ s ⟧⊢ = ⟦ f ⟧⊢ ˢ ⟦ s ⟧⊢
```

What coincidence does application require? The coincidence is of type *meaning*, not just of type *syntax*. If we have two candidates for S (functions in $[\![\Gamma]\!]^{\mathsf{C}} \to \mathsf{U}$), then they must be equal up to Agda's intensional equality of functions, which most certainly normalizes under binders. We have stolen the computational power we need from the host language!

Note, by the way, the care with which I have manipulated T in the rules for LAM and _$_. As Pierce and Turner [1998] observed, we should expect to push types into introduction forms and infer them from elimination forms. Correspondingly, I choose a *curried* T in the type of LAM, so that the type pushed in will yield a constraint

$$T\ \gamma\ s \equiv \ldots$$

amenable to pattern unification. A similar analysis leads to the curried choice in _$_, but this time we expect to learn T from synthesized type of the function. By aligning our definitions with the usual flow of type information, we turn Agda's constraint solver into a rather effective bidirectional type checker for KIPLING. Of course, we need to give top level types to our definitions, but we shall never need to supply these hidden shallow types manually when building terms.

This completes the presentation of KIPLING—its types, its type-safe terms, and its tagless interpreter. Rather, it would do, had I not made a rather subtle deliberate mistake.[5] We shall find out what this mistake is—and fix it—once we have tried to construct some KIPLING programs and proofs.

7. Programming in KIPLING (First Attempt)

If you can wait and not be tired by waiting, then you might enjoy using Agda as an interactive editor and evaluator for this implementation of KIPLING. Let us start with a little device to allow us to state types in KIPLING and recover the corresponding shallow type to use when building terms.

```
[_] : { T : ∀ {Γ} → ⟦ Γ ⟧C → U } →
      (∀ {Γ} → Γ ⋆ T) → Set
[_] { T } _ = ∀ {Γ} → Γ ⊢ T
```

Closed types should make sense in any context. If we write a closed KIPLING type, there is a good chance Agda can infer the shallow type (polymorphic in the unused context) which it represents. We may then use that polymorphic shallow type to compute the Agda

[4] It's frankly astonishing how effective Agda's implicit syntax mechanism turns out to be. The trouble is that the system's limits are far from clear. It is hard to tell what *shouldn't* work, and what is rather a lacuna.

[5] A mistake becomes deliberate when you decide to keep it in your paper.

type the corresponding KIPLING terms should inhabit if we are to be able to use them under binders.

Given primitive recursion for NAT, a reasonable first assignment is addition. Agda's interactive construction technology allows us to see what we are doing, although we see the interpreted types and values, rather than the KIPLING syntax. Even so, the pragmatic strategy is to develop examples with the shallow embedding, then translate them. Here, then, is addition, with de Bruijn indices decoded for human consumption in comments.[6]

```
ADD : [PI NAT (PI NAT NAT)]
ADD = LAM{-x-} (LAM{-y-} (REC (VAR{-x-}(pop top)) NAT
        VAR{-y-}top
        (LAM (LAM{-sum-} (SU VAR{-sum-}top)))
      ))
```

Let us confirm that two and two make four. If we ask Agda to evaluate, even without giving an environment, we find

```
    ⟦ ADD $ SU (SU ZE) $ SU (SU ZE) ⟧⊢
=   λ γ → su (su (su (su ze)))
```

which is four in any environment.

Programming addition is all very well, but we should really try to use KIPLING as a proof language, too. Let us start by reflecting the Booleans as triumph and disaster, respectively.

```
TRUE : ∀ {Γ} (b : Γ ⊢ κ '2') → Γ ⋆ _
TRUE b = IF b ONE ZERO
```

The _ in the type means 'go figure!', and Agda does indeed figure out which shallow type I mean from the KIPLING type I write. Note that TRUE is not a KIPLING function. Rather it is a schematic Agda abbreviation for KIPLING types. When used in KIPLING, it must always be fully applied.

What should we like to be true? We should be able to test, and hence assert, equality of numbers. Here is the code: I have the unfair advantage that my thesis explains the systematic translation of pattern matching to recursion operators. A machine can be persuaded to emit the following encoding:

```
NATEQ : [PI NAT (PI NAT TWO)]
NATEQ =
  LAM{-x-} (REC (VAR{-x-}top) (PI NAT TWO)
    (LAM{-y-} (REC (VAR{-y-}top) TWO
      TT                  -- NATEQ $ ZE $ ZE = TT
      (LAM (LAM FF))      -- NATEQ $ ZE $ SU _ = FF
    ))
    (LAM{-x-} (LAM{-xq-}
      (LAM{-y-} (REC (VAR{-y-}top) TWO
        FF  -- NATEQ $ SU x $ ZE = FF
        (LAM (LAM
          (VAR{-xq-}(pop (pop (pop (top))))
          $ VAR{-y-}(pop top))))
              -- NATEQ $ SU x $ SU y = NATEQ $ x $ y
      ))
    ))
  )
```

Defining equality in this manner gives us an easy proof that the constructors of NAT are injective and disjoint: by computation, TRUE (NATEQ $ SU x $ ZE) represents the same shallow type as ZERO; by computation, TRUE (NATEQ $ SU x $ SU y) means the same as TRUE (NATEQ $ x $ y). On the other hand, it takes work to show that this equality is *reflexive*. Let us begin. We may copy the goal into the type parameter of an induction—now we are using its bound variable, y below—and see if that makes progress.

```
REFL : [PI{-x-} NAT
         (TRUE (NATEQ $ VAR{-x-}top $ VAR{-x-}top))]
REFL =
  LAM{-x-} (REC{-y-} (VAR{-x-}top)
    (TRUE (NATEQ $ VAR{-y-}top $ VAR{-y-}top))
    ?
    ?
  )  -- does not typecheck, but it should
```

It is indeed a horror to watch the things you gave your life to, broken, but let us stoop to build them up with worn out tools. Agda gives a type error, reporting a conflict:

$$\Gamma, \kappa\,\text{`}\mathbb{N}\text{'} \neq \Gamma$$

How is it possible that we compare different contexts? Do we need to worry about weakening and substitution, after all? Has the whole business been a mirage? It is time to stop hacking and start thinking.

8. How does it work?

I claim to have implemented KIPLING, but I am apparently lying, as the failure of the above construction—a textbook mathematical induction—demonstrates. Let us classify what would constitute success.

I should, first of all, clarify the assumptions I make about $\equiv$ in Agda: it is not precisely specified, nor is it entirely clear what is implemented. However, Agda is broken if its definitional equality is not a congruence $\equiv$ which, moreover, validates the β-rule and the defining equations of programs with non-overlapping patterns. For convenience, I shall assume that Agda also supports the η-law for functions.

$$f \equiv \lambda\, x \to f\ x \qquad \text{if } x \notin f$$

We may thus consider all functions from environments to take the form $\lambda\ \gamma \to \ldots$ without further worry. I assume also that Agda's typing and equality rules are stable under well typed substitution. Again, if this somehow does not hold, the fault is not mine!

We should identify a syntactic translation $-^\dagger$ from KIPLING syntax on paper to terms in Agda. For types and terms, $T^\dagger$ and $t^\dagger$ are just the de Bruijn index translations of the syntax, implicitly depending on the variables in scope: the KIPLING grammar is scoped, after all, and the syntactic productions are mapped one-to-one. Note that de Bruijn index translation respects substitution, so $t[s]^\dagger = t^\dagger[s^\dagger]$.

We shall also need an *interpretation* $-^\ddagger$ from KIPLING syntax, taking contexts to Cx, types to functions from environments to U, and terms to functions from environments γ to some **El** $(T\ \gamma)$. I shall spell this translation out more fully.

$$\begin{aligned}
\mathcal{E}^\ddagger &= \text{`}\mathcal{E}\text{'} \\
(\Gamma, S)^\ddagger &= \Gamma^\ddagger, S^\ddagger \\
\text{ZERO}^\ddagger &= \kappa\,\text{`Zero'} \\
\text{ONE}^\ddagger &= \kappa\,\text{`1'} \\
\text{TWO}^\ddagger &= \kappa\,\text{`2'} \\
\text{NAT}^\ddagger &= \kappa\,\text{`}\mathbb{N}\text{'} \\
(\text{SG}\ S\ T)^\ddagger &= \kappa\,\text{`}\Sigma\text{'}\ \mathsf{s}\ S^\ddagger\ \mathsf{s}\ \mathsf{a}\ T^\ddagger \\
(\text{PI}\ S\ T)^\ddagger &= \kappa\,\text{`}\Pi\text{'}\ \mathsf{s}\ S^\ddagger\ \mathsf{s}\ \mathsf{a}\ T^\ddagger \\
(\text{IF}\ b\ T\ F)^\ddagger &= \kappa\,\text{`}\mathbf{If}\text{'}\ \mathsf{s}\ b^\ddagger\ \mathsf{s}\ T^\ddagger\ \mathsf{s}\ F^\ddagger \\
t^\ddagger &= [\![\, t^\dagger \,]\!]^\vdash
\end{aligned}$$

In order for computation to make sense, we had better be sure that interpretation respects substitution. Syntactic substitution in

[6] Comprehension of de Bruijn syntax is often proposed as a reverse Turing test. I sometimes find this worrying.

KIPLING should correspond to instantiating the environment after interpretation.

PROPOSITION 1 (Interpretation Respects Substitution). *The following implications hold.*

$$\begin{array}{rc} & \Gamma, x : S \vdash T\ \mathrm{TYPE} \ \wedge\ \Gamma \vdash s : S \ \wedge \\ & T^\ddagger\ :\ [\![\Gamma^\ddagger, S^\ddagger]\!]^C \to \mathsf{U} \ \wedge \\ & s^\ddagger\ :\ (\gamma\ :\ [\![\Gamma^\ddagger]\!]^C) \to \mathbf{El}\ (S^\ddagger\ \gamma) \\ \Rightarrow & (T[s])^\ddagger \equiv {}^\wedge T^\ddagger\ {}^\mathsf{s}\ s^\ddagger : [\![\Gamma^\ddagger]\!]^C \to \mathsf{U} \end{array}$$

$$\begin{array}{rc} & \Gamma, x : S \vdash t : T \ \wedge\ \Gamma \vdash s : S \ \wedge \\ & T^\ddagger\ :\ [\![\Gamma^\ddagger, S^\ddagger]\!]^C \to \mathsf{U} \ \wedge \\ & t^\ddagger\ :\ (\gamma\ :\ [\![\Gamma^\ddagger, S^\ddagger]\!]^C) \to \mathbf{El}\ (T^\ddagger\ \gamma) \ \wedge \\ & s^\ddagger\ :\ (\gamma\ :\ [\![\Gamma^\ddagger]\!]^C) \to \mathbf{El}\ (S^\ddagger\ \gamma) \\ \Rightarrow & (t[s])^\ddagger \equiv {}^\wedge t^\ddagger\ {}^\mathsf{s}\ s^\ddagger : (\gamma\ :\ [\![\Gamma^\ddagger]\!]^C) \to \mathbf{El}\ (T^\ddagger\ (\gamma, s^\ddagger\ \gamma)) \end{array}$$

Note that I have been careful to require not only that T, t, and s make sense according to the KIPLING rules, but also that their Agda translations make sense. We shall thus be safe to consider whether the interpretations do the right thing, without worrying about whether they are well typed. The type safety of the translation is the next concern.

We should lift our translation from syntax to judgments. For each KIPLING judgment, we should identify what we expect to be true in Agda about its translations. We should then attempt to prove that the Agda claims always hold, by mutual induction on KIPLING derivations. The cases which fail will reveal the bugs. Below, I have been careful to augment the direct interpretation of each judgment with some hygienic conditions about the well-formedness of the judgment in the first place, thus strengthening our inductive hypotheses.

PROPOSITION 2 (Translations Preserve Judgments). *The following implications all hold.*

$$\begin{array}{lll} \Gamma \vdash \mathrm{VALID} & \Rightarrow \Gamma^\ddagger\ :\ \mathsf{Cx} \\ \Gamma \vdash T\ \mathrm{TYPE} & \Rightarrow T^\dagger\ :\ \Gamma^\ddagger \star T^\ddagger\ \wedge\ \Gamma^\ddagger\ :\ \mathsf{Cx} \\ \Gamma \vdash t : T & \Rightarrow t^\dagger\ :\ \Gamma^\ddagger \vdash T^\ddagger\ \wedge \\ & \quad T^\dagger\ :\ \Gamma^\ddagger \star T^\ddagger\ \wedge\ \Gamma^\ddagger\ :\ \mathsf{Cx} \\ \Gamma \vdash S \equiv T & \Rightarrow S^\ddagger \equiv T^\ddagger : [\![\Gamma^\ddagger]\!]^C \to \mathsf{U}\ \wedge \\ & \quad S^\dagger\ :\ \Gamma^\ddagger \star S^\ddagger\ \wedge\ T^\dagger\ :\ \Gamma^\ddagger \star T^\ddagger\ \wedge \\ & \quad \Gamma^\ddagger\ :\ \mathsf{Cx} \\ \Gamma \vdash s \equiv t : T & \Rightarrow s^\ddagger\ \equiv\ t^\ddagger\ :\ (\gamma\ :\ [\![\Gamma^\ddagger]\!]^C) \to \mathbf{El}\ (T^\ddagger\ \gamma)\ \wedge \\ & \quad s^\dagger\ :\ \Gamma^\ddagger \vdash T^\ddagger\ \wedge\ t^\dagger\ :\ \Gamma^\ddagger \vdash T^\ddagger\ \wedge \\ & \quad T^\dagger\ :\ \Gamma^\ddagger \star T^\ddagger\ \wedge\ \Gamma^\ddagger\ :\ \mathsf{Cx} \end{array}$$

Suppose we can prove the former. Let us try to prove the latter.

Proof attempt for Proposition 2. We proceed by induction on KIPLING derivations.

For context validity, type formation, and the *syntax-directed* typing rules, the proof is by construction: each production in the grammar is mapped by $-^\dagger$ to a constructor whose type directly states the requirements for this proof to go through. For example, consider the PI case.

$$\frac{\Gamma \vdash S\ \mathrm{TYPE} \quad \Gamma, x : S \vdash T\ \mathrm{TYPE}}{\Gamma \vdash \mathrm{PI}\ x : S\ .\ T\ \mathrm{TYPE}}$$

Inductively, we have

$$S^\dagger : \Gamma^\ddagger \star S^\ddagger \qquad T^\dagger : \Gamma^\ddagger, S \star T^\ddagger \qquad \Gamma^\ddagger\ :\ \mathsf{Cx}$$

and we note the type of the PI constructor, suitably instantiated:

$$\mathrm{PI}\ :\ \Gamma^\ddagger \star S^\ddagger \to \Gamma^\ddagger, S^\ddagger \star T^\ddagger \to \Gamma^\ddagger \star {}^\kappa\text{`}\Pi\text{'}\ {}^\mathsf{s}\ S^\ddagger\ {}^\mathsf{s}\ {}^\wedge T^\ddagger$$

Hence

$$\mathrm{PI}\ S^\dagger\ T^\dagger\ :\ \Gamma^\ddagger \star {}^\kappa\text{`}\Pi\text{'}\ {}^\mathsf{s}\ S^\ddagger\ {}^\mathsf{s}\ {}^\wedge T^\ddagger \qquad \Gamma^\ddagger\ :\ \mathsf{Cx}$$

as required. Let us consider also application, where substitution is involved.

$$\frac{\Gamma \vdash f : \mathrm{PI}\ x : S\ .\ T \quad \Gamma \vdash s : S}{\Gamma \vdash f\ \$\ s : T[s]}$$

Inductively, we have some basic hygiene and

$$f^\dagger\ :\ \Gamma^\ddagger \vdash {}^\kappa\mathrm{PI}\ {}^\mathsf{s}\ S^\ddagger\ {}^\mathsf{s}\ {}^\wedge T^\ddagger \qquad s^\dagger\ :\ \Gamma^\ddagger \vdash S^\ddagger$$

so we get the demanded hygiene and

$$f^\dagger\ \$\ s^\dagger\ :\ \Gamma^\ddagger \vdash {}^\wedge T^\ddagger\ {}^\mathsf{s}\ s^\ddagger$$

hence by Proposition 1, $(T[s])^\ddagger \equiv {}^\wedge T^\ddagger\ {}^\mathsf{s}\ s^\ddagger$ as required.

There is no syntax associated with the conversion rule

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash S \equiv T}{\Gamma \vdash s : T}$$

so we had better check that $s^\dagger$ does indeed have both types required of it. Inductively, we have

$$s^\dagger : \Gamma^\ddagger \vdash S^\ddagger \qquad S^\ddagger \equiv T^\ddagger : [\![\Gamma^\ddagger]\!]^C \to \mathsf{U}$$

Agda's definitional equality is a congruence, so we note that

$$\Gamma^\ddagger \vdash S^\ddagger\ \equiv\ \Gamma^\ddagger \vdash T^\ddagger\ :\ \mathsf{Set} \quad \text{hence} \quad s^\dagger\ :\ \Gamma^\ddagger \vdash T^\ddagger$$

and the case goes through—the inductive hypothesis assures us that Agda's definitional equality includes KIPLING's equational theory.

So, we had better check the equality rules, in order to validate that inductive assurance. Equivalence closure follows from Agda's equivalence closure. For type equality, the structural rules go through by congruence of Agda's $\equiv$, and the two computation rules go through by definition of '**If**'.

For the terms, we can check the computation rules by running the interpreted programs. Let us check the β-rule, for example.

$$\frac{\Gamma \vdash \Gamma, x : S \vdash t : T \quad \Gamma \vdash s : S}{\Gamma \vdash (\mathrm{LAM}\ x\ .\ t)\ \$\ s \equiv t[s] : T[s]}$$

Translating the left-hand side, we get

$${}^\wedge [\![\, t^\dagger \,]\!]^\vdash\ {}^\mathsf{s}\ [\![\, s^\dagger \,]\!]^\vdash \qquad \text{i.e.} \qquad {}^\wedge t^\ddagger\ {}^\mathsf{s}\ s^\ddagger$$

which must equal $(t[s])^\ddagger$ by Proposition 1.

For the structural rules, our induction can only go through if our interpreter is *compositional*. Our inductive hypotheses give us equality of interpretation for corresponding components—this had better be sufficient, via congruence, to deliver equality of interpretation of the whole. For example, in the interpretation of pairing,

$$[\![\, s\ \&\ t \,]\!]^\vdash\ =\ \lambda\, \gamma \to [\![\, s \,]\!]^\vdash\, \gamma, [\![\, t \,]\!]^\vdash\, \gamma$$

you can see that the 'skeleton' of the right-hand side—the term outside the recursive $[\![\ {}^\mathsf{s}\ ]\!]^\vdash$ calls—does not depend on s or t. Correspondingly, if we know that $[\![\, s' \,]\!]^\vdash\ \equiv\ [\![\, s \,]\!]^\vdash$ and $[\![\, t' \,]\!]^\vdash\ \equiv\ [\![\, t \,]\!]^\vdash$, then congruence gives us that $[\![\, s\ \&\ t \,]\!]^\vdash\ \equiv\ [\![\, s\ \&\ t \,]\!]^\vdash$.

In effect, then, to be compositional, the interpreter must be given as the fold of an algebra on *values* unaccompanied by the syntax which produced them. If we check the skeletons, there is nothing obviously wrong—everything outside the recursive calls appears to be independent of unevaluated terms. But as Hoare observed, this is not to say that there is obviously nothing wrong. When interpreting REC, I wrote:

$$\begin{array}{l} [\![\, \mathrm{REC}\ \{P\}\ n\ \_\ z\ s \,]\!]^\vdash\ =\ \mathbf{recHelp}\ {}^\mathsf{s}\ [\![\, n \,]\!]^\vdash\ \mathbf{where} \\ \quad \mathbf{recHelp}\ :\ (\gamma\ :\ \_) \to (n\ :\ \mathbb{N}) \to \mathbf{El}\ (P\ (\gamma\,,\, n)) \\ \quad \mathbf{recHelp}\ \gamma\ \mathsf{ze}\ \ \ =\ [\![\, z \,]\!]^\vdash\, \gamma \\ \quad \mathbf{recHelp}\ \gamma\ (\mathsf{su}\ n)\ =\ [\![\, s \,]\!]^\vdash\, \gamma\ n\ (\mathbf{recHelp}\ \gamma\ n) \end{array}$$

which looks compositional. However, Agda supports helper functions by λ-lifting them [Johnsson 1985], so we really get this:

$$[\![\text{REC}\ \{P\}\ n\ \_\ z\ s]\!]^{\vdash} = \mathbf{recHelp}\ \Gamma\ P\ z\ s\ ^{\mathsf{s}}\ [\![n]\!]^{\vdash}\ \mathbf{where}$$
$$\mathbf{recHelp}\ \Gamma\ P\ z\ s\ \gamma\ \mathsf{ze} = [\![z]\!]^{\vdash}\ \gamma$$
$$\mathbf{recHelp}\ \Gamma\ P\ z\ s\ \gamma\ (\mathsf{su}\ n) =$$
$$[\![s]\!]^{\vdash}\ \gamma\ n\ (\mathbf{recHelp}\ \Gamma\ P\ z\ s\ \gamma\ n)$$

which leaves the *syntax* of z and s on display, and indeed, the particular context Γ in which the interpretation was constructed. The helper function for IF is similarly afflicted. ***Proof attempt failed.***

Getting back to our type error, we can now see how the induction predicate, interpreted under two binders, failed to match the range of the goal type, under one, with mismatching contexts present in the normal forms of our stuck conditional and recursor.

The remedy is, however, apparent. We must restore compositionality. If we still have **if** and **rec** available from our earlier experimental shallow embedding to Set, we can write

$$[\![\text{IF}\ \{P\}\ b\ \_\ t\ f]\!]^{\vdash} =$$
$$^{\mathsf{K}}\mathbf{if}\ ^{\mathsf{s}}\ [\![b]\!]^{\vdash}\ ^{\mathsf{s}}\ ^{\wedge}(\mathbf{El}\cdot P)\ ^{\mathsf{s}}\ [\![t]\!]^{\vdash}\ ^{\mathsf{s}}\ [\![f]\!]^{\vdash}$$
$$[\![\text{REC}\ \{P\}\ n\ \_\ z\ s]\!]^{\vdash} =$$
$$^{\mathsf{K}}\mathbf{rec}\ ^{\mathsf{s}}\ [\![n]\!]^{\vdash}\ ^{\mathsf{s}}\ ^{\wedge}(\mathbf{El}\cdot P)\ ^{\mathsf{s}}\ [\![z]\!]^{\vdash}\ ^{\mathsf{s}}\ [\![s]\!]^{\vdash}$$

and recover the property that the interpreter is a fold.

Proof of Proposition 2 resumed. Again, we proceed by induction on derivations. As before, the syntax-directed rules are respected, by construction and appeal to Proposition 1. As before, the conversion rule is respected by congruence, appealing to the respected equation. Type equality is respected by congruence for the structural rules, and by the program equations for '**If**'. The structural rules for term equality are now respected by a fully compositional interpreter, allowing appeal to congruence. The computation rules hold by definitional equality in Agda, and Proposition 1. □

Proposition 1 follows from a more general property, allowing *simultaneous substitution*.

PROPOSITION 3 (Respect for Simultaneous Substitution). *Let* Δ *and* Γ *be valid* KIPLING *contexts, with* $\Delta^{\ddagger}, \Gamma^{\ddagger} : \mathsf{Cx}$. *Let* σ *be a substitution from variables in* Δ *to terms over* Γ *such that*

$$\Gamma \vdash \sigma\ x\ :\ \sigma\ S \quad \textit{for}\ x{:}S \in \Delta$$

Moreover, let $\sigma^{\ddagger} : [\![\Gamma]\!]^{\mathsf{C}} \to [\![\Delta]\!]^{\mathsf{C}}$ *be an Agda function such that for each* $x{:}S \in \Delta$, *yielding* $x^{\ddagger} : \Delta^{\ddagger} \ni S^{\ddagger}$,

$$[\![x^{\ddagger}]\!]^{\ni} \cdot \sigma^{\ddagger} \equiv [\![(\sigma x)^{\ddagger}]\!]^{\vdash} : (\gamma\ :\ [\![\Gamma]\!]^{\mathsf{C}}) \to \mathbf{El}\ (S^{\ddagger}\ (\sigma^{\ddagger}\ \gamma))$$

Then, whenever $\Delta \vdash T\ \text{TYPE}$ $T^{\ddagger} : [\![\Delta^{\ddagger}]\!]^{\mathsf{C}} \to \mathsf{U}$
we have $(\sigma\, T)^{\ddagger} \equiv T^{\ddagger} \cdot \sigma^{\ddagger} : [\![\Gamma^{\ddagger}]\!]^{\mathsf{C}} \to \mathsf{U}$.
Moreover, if

$$\Delta \vdash t : T \quad T^{\ddagger} : [\![\Delta^{\ddagger}]\!]^{\mathsf{C}} \to \mathsf{U} \quad t^{\ddagger} : (\delta : [\![\Delta^{\ddagger}]\!]^{\mathsf{C}}) \to \mathbf{El}\ (T^{\ddagger}\ \delta)$$

then $(\sigma t)^{\ddagger} \equiv t^{\ddagger} \cdot \sigma^{\ddagger} : (\gamma\ :\ [\![\Gamma^{\ddagger}]\!]^{\mathsf{C}}) \to \mathbf{El}\ (T^{\ddagger}\ (\sigma^{\ddagger}\ \gamma))$

We may immediately note that substituting for the topmost free variable is a special case.

Proof of Proposition 1. When we have $\Gamma \vdash s : S$, with suitable translations to Agda, take

$$\Delta = \Gamma, x{:}S \quad \sigma = [s/x] \quad \sigma^{\ddagger}\ \gamma\ =\ \gamma, s^{\ddagger}\ \gamma$$

Note that for variable $x^{\dagger} = \mathsf{top}$,

$$[\![\mathsf{top}]\!]^{\ni}\ (\gamma, s^{\ddagger}\ \gamma)\ \equiv\ s^{\ddagger}\ \gamma$$

and for other variables $y^{\dagger} = \mathsf{pop}\ i$, with $\sigma\ y^{\dagger} = i$

$$[\![\mathsf{pop}\ i]\!]^{\ni}\ (\gamma, s^{\ddagger}\ \gamma)\ \equiv\ [\![i]\!]^{\ni}\ \gamma$$

Apply Proposition 3. □

Now let us prove the general case.

Proof of Proposition 3. We proceed by induction on the syntax of KIPLING. The Agda typing of the interpretations relies on the typing of their components, so inductive hypotheses are applicable. When we pass under a binder, moving from Δ to $\Delta, y : R$, we extend Γ to $\Gamma, y' : \sigma\ R$, lifting σ to $\tau = \sigma\,[y'/y]$, and lift $\sigma^{\ddagger}$ to $\tau^{\ddagger}$ where

$$\tau^{\ddagger}(\gamma, y) = (\tau^{\ddagger}\ \gamma, y)$$

preserving the required relationship with σ for top by construction and pop i by our assumption about $\sigma^{\ddagger}$.

As the translations are compositional and Agda's equality is a congruence, the conclusion for each type- and term-former follows structurally from the inductive hypotheses. The case for variables is exactly covered by our assumption about the relationship between σ and $\sigma^{\ddagger}$. □

The upshot is that with constructors reflecting the typing rules at the level of type meanings, and a compositional interpreter assigning term meanings, we acquire a translation from KIPLING to a type-safe syntax.

9. The Proof of the KIPLING Pudding

Now that we are sure the translation works, let us complete the proof that the equality test for NAT is reflexive. The construction is straightforward.

```
REFL : [PI_{-x-} NAT
         (TRUE (NATEQ $ VAR_{-x-} top $ VAR_{-x-} top))]
REFL =
  LAM_{-x-} (REC_{-y-} (VAR_{-x-} top)
    (TRUE (NATEQ $ VAR_{-y-} top $ VAR_{-y-} top))
    VOID  -- TRUE (NATEQ $ ZE $ ZE) = ONE
    (LAM_{-x-} (LAM_{-xq-} (VAR_{-xq-} top))))
        -- TRUE (NATEQ $ SU x $ SU x)
        -- = TRUE (NATEQ $ x $ x)
```

We may similarly observe that $x + 0 = x$.

```
ADDZE : [PI_{-x-} NAT
          (TRUE (NATEQ $ (ADD $ VAR_{-x-} top $ ZE)
                       $ VAR_{-x-} top))]
ADDZE =
  LAM_{-x-} (REC_{-y-} (VAR_{-x-} top)
    (TRUE (NATEQ $ (ADD $ VAR_{-y-} top $ ZE)
                 $ VAR_{-y-} top))
    VOID
    (LAM_{-x-} (LAM_{-xq-} (VAR_{-xq-} top))))
```

Finally, let me show that $x + \text{SU}\ y = \text{SU}\ x + y$.

```
ADDSU : [PI_{-x-} NAT (PI_{-y-} NAT (TRUE (NATEQ
          $ (ADD $ VAR_{-x-} (pop top) $ SU (VAR_{-y-} top))
          $ SU (ADD $ VAR_{-x-} (pop top) $ VAR_{-y-} top))))]
ADDSU =
  LAM_{-x-} (LAM_{-y-} (REC_{-z-} (VAR_{-x-} (pop top))
    (TRUE (NATEQ
      $ (ADD $ VAR_{-z-} top $ SU (VAR_{-y-} (pop top)))
      $ SU (ADD $ VAR_{-z-} top $ VAR_{-y-} (pop top))))
    (REFL $ SU (VAR_{-y-} top))
    (LAM_{-x-} (LAM_{-xq-} (VAR_{-xq-} top)))))
```

These proofs all rely on type-level computation, and they all check. There is a considerable performance penalty induced by the layer of translation, but the encoding demonstrably works.

10. Discussion

This paper is a contribution to the practice of programming with dependent types, allowing 'domain-specific language' techniques to be applied to more precise languages. Whilst the current implementation, available online via

```
darcs get http://personal.cis.strath.ac.uk/~conor/DepRep
```

is rather slow, the full power of partial evaluation in compilation stands at the ready [Brady and Hammond 2010].

It is also a technical contribution to the practice of formal metatheory for dependent type systems, in the tradition of Barras [1996, 1999]; Barras and Werner [1997]; McKinna and Pollack [1993, 1999]; Pollack [1994]. Where these formalisations start from untyped representations of dependently typed terms, we now have a technique for type-safe representation, in the manner of Danielsson [2006] and Chapman [2009], but with type equality silently resolved by the host language.

We have successfully encoded KIPLING, a minimal dependently typed language in a type safe manner and checked that all KIPLING judgments hold in translation. The key ingredients were

1. a universe U of Agda types corresponding to canonical KIPLING types, defined by induction-recursion, just as Dybjer and Setzer [1999] taught us, allowing interpretation of KIPLING types $\Gamma \vdash T$ TYPE as 'shallow types'—Agda functions $T^\ddagger : [\![\Gamma^\ddagger]\!]^{\mathsf{C}} \to \mathsf{U}$;
2. de Bruijn indices, just as Altenkirch and Reus [1999] taught us, indexed by shallow types—weakening is just $\cdot\,\mathbf{fst}$;
3. a first-order deep embedding of KIPLING types, indexed to express *representability* of shallow types, just as Crary et al. [1998] taught us;
4. mutually defined with the latter, a first-order syntax of 'deep' terms, indexed by shallow types, so that convertability of deep types is supplanted by *coincidence* of shallow types—that is what I am teaching;
5. a tagless interpreter, as Augustsson and Carlsson [1999] taught us, taking deep terms $\Gamma \vdash t : T$ to their shallow counterparts $[\![t^\dagger]\!]^\vdash : (\gamma : [\![\Gamma^\ddagger]\!]^{\mathsf{C}}) \to \mathbf{El}\ (T^\ddagger\ \gamma)$, used crucially to *construct* shallow dependent types.

The mutual definition of types, terms and their values, by indexed induction-recursion [Dybjer and Setzer 2001], reflects exactly the presence of types in terms and terms computing in types that we need to define dependently typed calculi.

The technique is thus adaptable to a variety of languages, but with at least one note of caution: we are at the mercy of the definitional equality in the host language. On the one hand, we cannot expect to model a theory—the proof-irrelevant Observational Type Theory [Altenkirch et al. 2007], for example—whose definitional equality is strictly (or rather, lazily) more generous than Agda's. On the other hand, we are obliged to accept the generosity of our host, whether we like it or not! For example, the η-law holds for KIPLING functions, without us asking for it.

```
F1 : [PI (PI NAT NAT) (PI NAT NAT)]
F1 = LAM (LAM (VAR (pop top) $ VAR top))
F2 : [PI (PI NAT NAT) (PI NAT NAT)]
F2 = LAM (VAR top)
FEQ : [[F1 {'E'}]]⊢ ≡ [[F2 {'E'}]]⊢
FEQ = refl
```

Another less than ideal aspect of this technique is that, while we have an interpreter, we have no means to recover a syntactic presentation of value: we do not have the *normalization* function. Modifying the universe construction to admit free variables in the interpretation of types might provide the means to step up from evaluation to strong normalization: this is an active topic of further research. Given a normalization function, we may become free to choose the definitional equality that we really want for our object languages, provided we can show that syntactic equality of types' normal forms ensures an isomorphism between them.

Such considerations aside, it is pleasing at last to seize the chance to model parts of our own languages, as we have modelled simply typed 'clients' in the past. We now have a new reflective technology, a new opportunity for generic programming with dependent types. If you can dream, and not make dreams your master, the prospects for progress are exceedingly good.

References

T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In J. Flum and M. Rodríguez-Artalejo, editors, *CSL*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. ISBN 3-540-66536-6.

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In A. Stump and H. Xi, editors, *PLPV*, pages 57–68. ACM, 2007. ISBN 978-1-59593-677-6.

L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter. Available at `http://www.cs.chalmers.se/ augustss/cayenne/interp.ps`, 1999.

A. I. Baars and S. D. Swierstra. Type-safe, self inspecting code. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM. ISBN 1-58113-850-4. doi: http://doi.acm.org/10.1145/1017472.1017485.

B. Barras. Coq en coq. Rapport de Recherche 3026, INRIA, Oct. 1996.

B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, Nov. 1999.

B. Barras and B. Werner. Coq in coq. `http://pauillac.inria.fr/ barras/coqincoq.ps.gz`, 1997.

E. Brady and K. Hammond. A verified staged interpreter is a verified compiler. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *GPCE*, pages 111–120. ACM, 2006. ISBN 1-59593-237-2.

E. Brady and K. Hammond. Scrapping your Inefficient Engine: using Partial Evaluation to Improve Domain-Specific Language Implementation,. In *ICFP 2010*. ACM, 2010. To appear.

J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

J. Chapman. Type theory should eat itself. *Electr. Notes Theor. Comput. Sci.*, 228:21–36, 2009.

J. Chapman. *Type checking and normalisation*. PhD thesis, University of Nottingham, 2008.

J. Chapman, P. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *ICFP 2010*. ACM, 2010. To appear.

C. Chen and H. Xi. Meta-programming through typeful code representation. In C. Runciman and O. Shivers, editors, *ICFP*, pages 275–286. ACM, 2003. ISBN 1-58113-756-7.

K. Crary, S. Weirich, and J. G. Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP*, pages 301–312, 1998.

H. B. Curry and R. Feys. *Combinatory Logic Volume I*. Amsterdam, 1958.

L. Damas and R. Milner. Principal type-schemes for functional programming languages. In *Ninth (Albuquerque, NM)*, pages 207–212. ACM, January 1982.

N. A. Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In T. Altenkirch and C. McBride, editors, *TYPES*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2006. ISBN 978-3-540-74463-4.

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

N. G. de Bruijn. Telescopic Mappings in Typed Lambda-Calculus. *Information and Computation*, 91:189–204, 1991.

P. Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. CUP, 1991.

P. Dybjer and A. Setzer. Indexed induction-recursion. In R. Kahle, P. Schroeder-Heister, and R. F. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 93–113. Springer, 2001. ISBN 3-540-42752-X.

P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *TLCA*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 1999. ISBN 3-540-65763-0.

F. Forsberg and A. Setzer. Inductive-inductive definitions. 10pp. Submitted to LICS 2010, 2010.

R. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

G. Huet and G. Plotkin, editors. *Electronic Proceedings of the First Annual BRA Workshop on Logical Frameworks (Antibes, France)*, 1990.

G. P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 190–203, 1985.

P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.

C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

J. McKinna and R. Pollack. Pure type systems formalized. In M. Bezem and J. F. Groote, editors, *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 289–305. Springer, 1993. ISBN 3-540-56517-5.

J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reasoning*, 23(3-4):373–409, 1999.

D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.

U. Norell. Dependently typed programming in agda. In P. W. M. Koopman, R. Plasmeijer, and S. D. Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008. ISBN 978-3-642-04651-3.

U. Norell. *Towards a Practical Programming Language based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

E. Palmgren. On universes in type theory. In *Proceedings of the meeting Twenty-five years of constructive type theory*. Oxford University Press, 1998.

E. Pasalic, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *ICFP*, pages 218–229, 2002.

B. C. Pierce and D. N. Turner. Local type inference. In *POPL*, pages 252–265, 1998.

R. Pollack. Dependently typed records in type theory. *Formal Asp. Comput.*, 13(3-5):386–402, 2002.

R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, Univ. of Edinburgh, 1994. URL `http://homepages.inf.ed.ac.uk/rpollack/export/thesis.ps.gz`.

R. Pollack. Implicit syntax. URL `ftp://ftp.dcs.ed.ac.uk/pub/lego/ImplicitSyntax.ps.Z`. An earlier version of this paper appeared in [Huet and Plotkin 1990], 1992.

A. Coda

With a little space left, let me show you a universe construction I learned from Palmgren [1998], with thanks to Peter Hancock.

Give our existing universe, U, we may make another, very slightly larger.

```
data Û : Set where
  'U'  : Û
  'El' : U → Û

Êl : Û → Set
Êl 'U'       = U
Êl ('El' T)  = El T
```

The $\hat{\mathsf{U}}$ universe tops up the types from U with a code for U itself. We can repeat the KIPLING construction, taking $\hat{\mathsf{U}}$ as the underlying universe for contexts and shallow types, and now we may write:

```
mutual
  data _⋆_ (Γ : Cx) : (⟦ Γ ⟧ᶜ → Û) → Set where
    SET  : Γ ⋆ᴷ 'U'
    EL   : (T : Γ ⊢ᴷ 'U') → Γ ⋆ᴷ 'El' ˢ ⟦ T ⟧⊢
    ZERO : Γ ⋆ᴷ ('El' 'Zero')
    ONE  : Γ ⋆ᴷ ('El' '1')
    ...
```

I have added a type of SETs, and the means to interpret terms of that type as types in the object language. I modify the term language, accordingly. We shall need to write types as terms, now.

```
TY : ∀ {T} → Γ ⋆ᴷ 'El' ˢ T → Γ ⊢ᴷ 'U'
⟦ TY {T} _ ⟧⊢ = T
```

Moreover, I adjust REC to operate on the larger universe, allowing us to compute values in SET. To do so, I must use the variable binding technique, as $\hat{\mathsf{U}}$ contains U, but not $\mathbb{N} \to \mathsf{U}$, for example.

```
REC : ∀ {P}
  → (n : Γ ⊢ᴷ ('El' 'ℕ'))
  → Γ, ᴷ('El' 'ℕ') ⋆ P
  → Γ ⊢ᴬ P ˢ ᴷze
  → (Γ, ᴷ('El' 'ℕ')), P ⊢ⱽ (ⱽλ γ n h → P (γ, su n))
  → Γ ⊢ᴬ P ˢ ⟦ n ⟧⊢

⟦ REC {P} n _ z s ⟧⊢ = ᴷrec ˢ ⟦ n ⟧⊢ ˢ ᴬ(Êl · P)
  ˢ ⟦ z ⟧⊢ ˢ λ γ n h → ⟦ s ⟧⊢ ((γ, n), h)
```

In an unforgiving minute, we have run far from KIPLING and the power of IF alone. We may now, for example, define vector-like structures—here, just vectors of numbers.

```
VEC : ∀ {Γ} → (n : Γ ⊢ᴷ ('El' 'ℕ')) → Γ ⋆ _
VEC n = EL (REC n SET
  (TY ONE)
  (TY (SG NAT (EL (VAR (pop top))))))
```

Here, for example, is the function which computes the decreasing sequence of the numbers below its input.

```
COUNTDOWN : [PI NAT (VEC (VAR top))]
COUNTDOWN = LAM (REC (VAR top) (VEC (VAR top))
  VOID
  (VAR (pop top) & VAR top))
```

And so, let us blast off into the universe…