Dependent types:Π,Σ, ground families, paths

July 28, 2017

Judgement forms

The core judgement forms of a type theory are

$$\alpha : \mathbf{ty}$$
$$a : \mathbf{el}\ \alpha$$

and where necessary[1] the equational forms

$$\alpha = \alpha' : \mathbf{ty}$$
$$a = a' : \mathbf{el}\ \alpha$$

Around this we have the apparatus of contexts and substitutions. The metamathematical semantics of a type theory is (for me) most conveniently expressed (*a la* Peter Dybjer) as a functor $[\_] : C^{\mathbf{op}} \to Fam(Set)$, where C is a category whose objects are called contexts, and Set is the category of sets. The functor assigns to each context an ordered pair $\mathbf{ty}, \mathbf{tm}$, in which $\mathbf{ty}$ is a set whose elements model types, and $\mathbf{tm}$ assigns to each element of $\mathbf{ty}$ the set whose elements model data of that type. There is a certain 'comprehension' structure, whereby contexts can be extended by introducing a new variable of a certain type. We are going to change this slightly, so that the codomain of the semantic functor will not be $Fam(Set)$, but rather $Fam(Set^2)$. Furthermore, context extension and comprehension will involve names.

The type theory in the next few pages derives, via my errors and insights, from Martin-Löf, as of sometime in the 90's. I learnt about it primarily from a paper by Daniel Fridlender [1], also from various writings of Alvaro Tasistro [2], and I think even from lecture notes in Per's handwriting emanating from Gothenburg. It is a pity this type theory is not better known. Not only does it have subtle and interesting (if confusing) features, but it is actually quite workable[2] in practice.

Some of the key features.

Explicit substitutions, thought of as maps between contexts. This is not particularly radical nowadays, though seldom is this combined with named variables (rather than 'stack notation': as with Peter Dybjer's categories with families, with its $p : (\Gamma, \alpha) \to \Gamma$) and $q : \mathbf{el}\,\alpha\,p\,\{\Gamma, \alpha\}$).

Named variables, subcontexts and thinning.

Real life use of type theory by a human being necessitates the use of mnemonic names for variables. Perhaps a computer can identify variables by the position they occupy in a machine's stack, but human beings use associative lookup.

A context is a map with finite domain, associating types with names. It must be well-formed in a certain sense. A subcontext of context Δ is a restriction of Δ (to a subset of the names) that makes sense on its own. We write this:

$$\Gamma \sqsubseteq \Delta$$

A substitution is also a map with finite domain, this time associating terms with names, that again is well-formed in a certain sense. Whether substitutions are equal depends on what domain and codomain you are considering.

The impact of named variables is largely in the area of 'thinning'.

Slices, index dependent families.

In a given context, there is not just the data about which you can say things, but there are also the things you can say about that data.

There is a new judgement form

$$\beta : \mathbf{ty}/\,\alpha$$

which says that β is something predicable of data of type α, or a type family indexed by α. One can think, approximately, of what is on the right-hand side as a very basic form of the arrow type, having a fixed

[1] In many type theories, the equational theory of types is trivial: there may be just one type, or the types can be given in advance of the terms. We care only about the typed equations. The notion of type equality is particularly important when type expressions can contain term expressions, as in dependent type theories.

[2] It is expressed here in a very noisy notation, so as not to use any overloaded operators – in real life one would use simply juxtaposition for all the various forms of 'application'.

codomain. By design, this arrow type cannot be iterated. (Compare with class notation and schemas in set-theory.) Given a type family β over α and an instance a of α, we index β with a, and form $\beta[a]$. Indexing is akin to application[3]. Correspondingly, type families can be formed by abstracting over a variable in the context used in a type expression – and who knows by what other means. This is akin to functional abstraction. However **ty** is not a type, but a mark of a pair of judgement forms. It is not the codomain of any function. Inasmuch as families resemble functions, they are a very simple form of function, with arguments always instances of types, never families over types.

Remark: the apparatus dealing with families is a little bit like the apparatus of classes that goes along with $\in$-theoretic set theory.

Putting things together, there are 10 (yes, 10!) forms of judgement, as follows.

$\Gamma : \mathbf{cxt}$	$\Gamma \sqsubseteq \Delta$
$\gamma : \Gamma\,\{\Delta\}$	$\gamma = \gamma' : \Gamma\,\{\Delta\}$
$\alpha : \mathbf{ty}$	$\alpha = \alpha' : \mathbf{ty}$
$a : \mathbf{el}\ \alpha$	$a = a' : \mathbf{el}\ \alpha$
$\beta : \mathbf{ty}/\,\alpha$	$\beta = \beta' : \mathbf{ty}/\,\alpha$

The top 4 judgment codes are really part of the general apparatus of contexts and substitutions with named variables and components.

The last 6 judgment forms are in 'categorical' form. Each of them, say J, has additionally a 'hypothetical' form: $J\ \{\Delta\}$. (One can perhaps think of $\gamma : \Gamma\,\{\Delta\}$ as a hypothetical judgement $\gamma :: \Gamma\,\{\Delta\}$, where the categorical part is a 'vectorised' or 'compound' form of $a : \alpha$.

Suggested readings (how to vocalise these things) and a few remarks:

$\Gamma : \mathbf{cxt}$	Γ is a context. (This expression) Γ *denotes* a context. A similar linguistic-semantic reading is possible for all judgement forms.
$\Gamma \sqsubseteq \Delta$	Γ is a subcontext or restriction of Δ. Δ is an extension or thinning of Γ.
$\gamma : \Gamma\,\{\Delta\}$	γ is a substitution or context-morphism from Δ to Γ. γ implements Γ, in Δ. Perhaps, γ is a valuation for Γ, in Δ.
$\gamma = \gamma' : \Gamma\,\{\Delta\}$	γ and γ' are the same substitution from Δ to Δ.
$\alpha : \mathbf{ty}\,\{\Delta\}$	α is/denotes a type, in Δ.
$\alpha = \alpha' : \mathbf{ty}\,\{\Delta\}$	α and α' are the same type, or synonymous as types in Δ.
$a : \mathbf{el}\,\alpha\,\{\Delta\}$	a is data, or an instance, or an object of/in type α, in context Δ.
$a = a' : \mathbf{el}\,\alpha\,\{\Delta\}$	a and a' are the same datum/instance/object of/in type α, in context Δ.
$\beta : \mathbf{ty}/\,\alpha\,\{\Delta\}$	β is a type-valued family or type-family indexed over, or by instances of type α, in context Δ. β is a slice or fibration over α, in context Δ.
$\beta = \beta' : \mathbf{ty}/\,\alpha\,\{\Delta\}$	β and β' are equal as type-families indexed over α, in context Δ.

I think I heard it said that the system described here (or some other close relative from the 90's derived from talks or notes by Martin-Löf on the topic of substitution calculus) may not be closed under α-conversion. I don't know if this is so, or even what exactly is meant, but it sounds worrisome.

I think I heard it said that further issues arise if one tries incorporating smoothly-working amenities for records with named fields (a topic investigated by Tasistro).

I have no real idea as to what particular versions of this system, real or imagined, these rumours pertain.

The little red numbers refer to the figures in Daniel Fridlender's paper [1]. (Another relevant source is Tasistro's [2], that may be his thesis, or part of it.) The little blue labels are mnemonics for myself. It challenging to try to present things in perfect conceptual order.

References

[1] Daniel Fridlender. A proof-irrelevant model of Martin-Löf's logical framework. *Mathematical Structures in Computer Science*, 12(6):771–795, 2002.

[3] We might think of $\beta[a]$ as a sentence in subject-predicate form

[2] Alvaro Tasistro. Formulation of Martin-Löf's theory of types with explicit substitutions, 1993.

contexts

formation

$$\text{F:04.01}\ \frac{}{() : \mathbf{cxt}} \qquad \text{F:04.02}\ \frac{\Gamma : \mathbf{cxt} \quad \alpha : \mathbf{ty}\,\{\Gamma\}}{(\Gamma, x : \alpha) : \mathbf{cxt}}\ (x\ \Gamma\text{-fresh})$$

substitution forms

$$\text{F:06.01}\mathbf{id}\ \frac{\Gamma : \mathbf{cxt}}{() : \Gamma\,\{\Gamma\}} \qquad \text{F:06.01}\mathbf{id}_c\ \frac{\Gamma : \mathbf{cxt}}{() = () : \Gamma\,\{\Gamma\}}$$

$$\text{F:06.03}\mathbf{comp}\ \frac{\gamma : \Gamma\,\{\Delta\} \quad \delta : \Delta\,\{\Theta\}}{\gamma\delta : \Gamma\,\{\Theta\}} \qquad \text{F:11.05}\mathbf{comp}_c\ \frac{\gamma = \gamma' : \Gamma\,\{\Delta\} \quad \delta = \delta' : \Delta\,\{\Theta\}}{\gamma\delta = \gamma'\delta' : \Gamma\,\{\Theta\}}$$

With global premises $\Gamma : \mathbf{cxt} \quad \alpha : \mathbf{ty}\,\{\Gamma\}$, and side condition ($x$ Γ-fresh):

$$\text{F:06.02}\mathbf{upd}\ \frac{\gamma : \Gamma\,\{\Delta\} \quad a : \mathbf{el}\,\alpha\gamma\,\{\Delta\}}{(\gamma, x \leftarrow a) : (\Gamma, x : \alpha)\,\{\Delta\}} \qquad \text{F:06.02}\mathbf{upd}_c\ \frac{\gamma = \gamma' : \Gamma\,\{\Delta\} \quad a = a' : \mathbf{el}\,\alpha\gamma\,\{\Delta\}}{(\gamma, x \leftarrow a) = (\gamma', x \leftarrow a') : (\Gamma, x : \alpha)\,\{\Delta\}}$$

equality and composition of substitutions

$$\text{F:11.07}\mathbf{id}_R\ \frac{\gamma : \Gamma\,\{\Delta\}}{\gamma() = \gamma : \Gamma\,\{\Delta\}} \qquad \text{F:11.06}\mathbf{assoc}\ \frac{\gamma : \Gamma\,\{\Delta\} \quad \delta : \Delta\,\{\Theta\} \quad \theta : \Theta\,\{\Phi\}}{\gamma(\delta\theta) = (\gamma\delta)\theta : \Gamma\,\{\Phi\}}$$

$$\text{F:11.08}\mathbf{id}_L\ \frac{\gamma : \Gamma\,\{\Delta\}}{()\gamma = \gamma : \Gamma\,\{\Delta\}} \qquad \text{F:11.09}\ \frac{\gamma : \Gamma\,\{\Delta\} \quad a : \mathbf{el}\,\alpha\gamma\,\{\Delta\} \quad \delta : \Delta\,\{\Theta\}}{(\gamma, x \leftarrow a)\delta = (\gamma\delta, x \leftarrow a\delta) : (\Gamma, x : \alpha)\,\{\Theta\}}\ (x\ \Gamma\text{-fresh})$$

$$\text{F:11.10}\ \frac{\gamma : \Gamma\,\{\Delta\} \quad a : \mathbf{el}\,\alpha\gamma\,\{\Delta\}}{(\gamma, x \leftarrow a) = \gamma : \Gamma\,\{\Delta\}}\ (x\ \Gamma\text{-fresh})$$

$$\text{F:11.01}\ \frac{\gamma : ()\,\{\Delta\} \quad \gamma' : ()\,\{\Delta\}}{\gamma = \gamma' : ()\,\{\Delta\}} \qquad \text{F:11.02}\ \frac{\gamma = \gamma' : \Gamma\,\{\Delta\} \quad x\gamma = x\gamma' : \mathbf{el}\,\alpha\gamma\,\{\Delta\}}{\gamma = \gamma' : (\Gamma, x : \alpha)\,\{\Delta\}}\ (x\ \Gamma\text{-fresh})$$

type equality

$$\text{F:05.03}\ \frac{a : \mathbf{el}\,\alpha\,\{\Gamma\} \quad \alpha = \alpha' : \mathbf{ty}\,\{\Gamma\}}{a : \mathbf{el}\,\alpha'\,\{\Gamma\}} \qquad \text{F:10.02}\ \frac{a = a' : \mathbf{el}\,\alpha\,\{\Gamma\} \quad \alpha = \alpha' : \mathbf{ty}\,\{\Gamma\}}{a = a' : \mathbf{el}\,\alpha'\,\{\Gamma\}}$$

$$\text{F:08.03}\ \frac{\beta : \mathbf{ty}/\,\alpha\,\{\Gamma\} \quad \alpha = \alpha' : \mathbf{ty}\,\{\Gamma\}}{\beta : \mathbf{ty}/\,\alpha'\,\{\Gamma\}} \qquad \text{F:13.02}\ \frac{\beta = \beta' : \mathbf{ty}/\,\alpha\,\{\Gamma\} \quad \alpha = \alpha' : \mathbf{ty}\,\{\Gamma\}}{\beta = \beta' : \mathbf{ty}/\,\alpha'\,\{\Gamma\}}$$

context thinning

$$\text{F:09.01}\ \frac{\Gamma : \mathbf{cxt}}{() \sqsubseteq \Gamma} \qquad \text{F:09.02}\ \frac{\Gamma \sqsubseteq \Delta \quad \alpha : \mathbf{ty}\,\{\Gamma\} \quad x : \mathbf{el}\,\alpha\,\{\Delta\}}{(\Gamma, x : \alpha) \sqsubseteq \Delta}\ (x\ \Gamma\text{-fresh})$$

These rules say how we can establish that one context is a *part* of another.

$$\text{F:07.02}\ \frac{\alpha : \mathbf{ty}\,\{\Gamma\} \quad \Gamma \sqsubseteq \Delta}{\alpha : \mathbf{ty}\,\{\Delta\}} \qquad \text{F:12.01}\ \frac{\alpha = \alpha' : \mathbf{ty}\,\{\Gamma\} \quad \Gamma \sqsubseteq \Delta}{\alpha = \alpha' : \mathbf{ty}\,\{\Delta\}}$$

$$\text{F:05.02}\ \frac{a : \mathbf{el}\,\alpha\,\{\Gamma\} \quad \Gamma \sqsubseteq \Delta}{a : \mathbf{el}\,\alpha\,\{\Delta\}} \qquad \text{F:10.01}\ \frac{a = a' : \mathbf{el}\,\alpha\,\{\Gamma\} \quad \Gamma \sqsubseteq \Delta}{a = a' : \mathbf{el}\,\alpha\,\{\Delta\}}$$

$$\text{F:08.02}\ \frac{\beta : \mathbf{ty}/\alpha\,\{\Gamma\} \quad \Gamma \sqsubseteq \Delta}{\beta : \mathbf{ty}/\alpha\,\{\Delta\}} \qquad \text{F:13.01}\ \frac{\beta = \beta' : \mathbf{ty}/\alpha\,\{\Gamma\} \quad \Gamma \sqsubseteq \Delta}{\beta = \beta' : \mathbf{ty}/\alpha\,\{\Delta\}}$$

$$\text{F:06.04}\mathbf{s\text{-}th}\ \frac{\theta : \Theta\,\{\Gamma\} \quad \Gamma \sqsubseteq \Delta}{\theta : \Theta\,\{\Delta\}} \qquad \text{F:11.03}\mathbf{s\text{-}th}_e\ \frac{\theta = \theta' : \Theta\,\{\Gamma\} \quad \Gamma \sqsubseteq \Delta}{\theta = \theta' : \Theta\,\{\Delta\}}$$

These rules pertain to basic judgements, and say that to establish J in a context, it is enough to establish J in a part of that context. (Hence: weakening or thinning.)

$$\text{F:06.05}\mathbf{t\text{-}th}\ \frac{\Gamma \sqsubseteq \Delta \quad \gamma : \Delta\,\{\Theta\}}{\gamma : \Gamma\,\{\Theta\}} \qquad \text{F:11.04}\mathbf{t\text{-}th}_e\ \frac{\Gamma \sqsubseteq \Delta \quad \gamma = \gamma' : \Delta\,\{\Theta\}}{\gamma = \gamma' : \Gamma\,\{\Theta\}}$$

These rules are concerned with hiding part of the codomain of a substitution.

variables

$$\text{F:05.01}\ \frac{\Gamma : \mathbf{cxt}}{x : \mathbf{el}\,\alpha\,\{\Gamma\}}\ (x : \alpha \text{ in } \Gamma) \qquad \text{F:10.06}\ \frac{\gamma : \Gamma\,\{\Delta\} \quad \alpha : \mathbf{ty}\,\{\Gamma\} \quad a : \mathbf{el}\,\alpha\gamma\,\{\Delta\}}{x(\gamma, y \leftarrow a) = a : \mathbf{el}\,\alpha\gamma\,\{\Delta\}}\ (x = y,\ x : \alpha \text{ in } \Gamma)$$

$$\text{F:10.06}\ \frac{\gamma : \Gamma\,\{\Delta\} \quad \alpha : \mathbf{ty}\,\{\Gamma\} \quad a : \mathbf{el}\,\alpha\gamma\,\{\Gamma\} \quad x\gamma = b : \mathbf{el}\,\beta\,\{\Delta\}}{x(\gamma, y \leftarrow a) = b : \mathbf{el}\,\beta\,\{\Delta\}}\ (x \neq y)$$

We might expect $x() = x : \mathbf{el}\,\alpha = x$ here, but it is not specific to variables:

$$\text{F:12.03}\ \frac{\alpha : \mathbf{ty}\,\{\Gamma\} \quad \gamma : \Gamma\,\{\Delta\} \quad \delta : \Delta\,\{\Theta\}}{\alpha(\gamma\delta) = (\alpha\gamma)\delta : \mathbf{ty}\ [\Theta]} \qquad \text{F:12.04}\ \frac{\alpha : \mathbf{ty}\,\{\Gamma\}}{\alpha() = \alpha : \mathbf{ty}\,\{\Gamma\}}$$

$$\text{F:10.04}\ \frac{a : \mathbf{el}\,\alpha\,\{\Gamma\} \quad \gamma : \Gamma\,\{\Delta\} \quad \delta : \Delta\,\{\Theta\}}{a(\gamma\delta) = (a\gamma)\delta : \mathbf{el}\,\alpha(\gamma\delta)\ [\Theta]} \qquad \text{F:10.05}\ \frac{a : \mathbf{el}\,\alpha\,\{\Gamma\}}{a() = a : \mathbf{el}\,\alpha\,\{\Gamma\}}$$

$$\text{F:13.04}\ \frac{\beta : \mathbf{ty}/\alpha\,\{\Gamma\} \quad \gamma : \Gamma\,\{\Delta\} \quad \delta : \Delta\,\{\Theta\}}{\beta(\gamma\delta) = (\beta\gamma)\delta : \mathbf{ty}/\alpha(\gamma\delta)\ [\Theta]} \qquad \text{F:13.04}\ \frac{\beta : \mathbf{ty}/\alpha\,\{\Gamma\}}{\beta() = \beta : \mathbf{ty}/\alpha\,\{\Gamma\}}$$

CF: 11.06 (assoc of subst. comp.), and 11.07 (nil subst. is right unit comp.)

substitution in judgements

$$\text{F:07.02}\ \frac{\alpha : \mathbf{ty}\,\{\Gamma\} \quad \gamma : \Gamma\,\{\Delta\}}{\alpha\gamma : \mathbf{ty}\,\{\Delta\}} \qquad \text{F:12.02}\ \frac{\alpha = \alpha' : \mathbf{ty}\,\{\Gamma\} \quad \gamma = \gamma' : \Gamma\,\{\Delta\}}{\alpha\gamma = \alpha'\gamma' : \mathbf{ty}\,\{\Delta\}}$$

$$\text{F:05.04}\ \frac{a : \mathbf{el}\,\alpha\,\{\Gamma\} \quad \gamma : \Gamma\,\{\Delta\}}{a\gamma : \mathbf{el}\,\alpha\gamma\,\{\Delta\}} \qquad \text{F:10.03}\ \frac{a = a' : \mathbf{el}\,\alpha\,\{\Gamma\} \quad \gamma = \gamma' : \Gamma\,\{\Delta\}}{a\gamma = a'\gamma' : \mathbf{el}\,\alpha\gamma\,\{\Delta\}}$$

$$\text{F:08.04}\ \frac{\beta : \mathbf{ty}/\alpha\,\{\Gamma\} \quad \gamma : \Gamma\,\{\Delta\}}{\beta\gamma : \mathbf{ty}/\alpha\gamma\,\{\Delta\}} \qquad \text{F:13.03}\ \frac{\beta = \beta' : \mathbf{ty}/\alpha\,\{\Gamma\} \quad \gamma = \gamma' : \Gamma\,\{\Delta\}}{\beta\gamma = \beta'\gamma' : \mathbf{ty}/\alpha\gamma\,\{\Delta\}}$$

type-families (slices/fibres/..), indexing, abstraction

Global premises: $\Gamma : \mathbf{cxt}, \quad \alpha : \mathbf{ty}\,\{\Gamma\}$

$$\text{F:07.04}\mathbf{slapp}\ \frac{\beta : \mathbf{ty}/\,\alpha\,\{\Gamma\} \quad a : \mathbf{el}\,\alpha\,\{\Gamma\}}{\beta[a] : \mathbf{ty}\,\{\Gamma\}}$$

$$\text{F:12.05}\mathbf{slapp}_c\ \frac{\beta = \beta' : \mathbf{ty}/\,\alpha\,\{\Gamma\} \quad a = a' : \mathbf{el}\,\alpha\,\{\Gamma\}}{\beta[a] = \beta'[a'] : \mathbf{ty}\,\{\Gamma\}} \qquad \text{F:12.06}\mathbf{slapp}_s\ \frac{\beta : \mathbf{ty}/\,\alpha\,\{\Gamma\} \quad a : \mathbf{el}\,\alpha\,\{\Gamma\} \quad \gamma : \Gamma\,\{\Delta\}}{(\beta[a])\gamma = (\beta\gamma)[a\gamma] : \mathbf{ty}\,\{\Delta\}}$$

Global side-condition:(x Γ-fresh)

$$\text{F:13.06}\zeta\ \frac{\beta, \beta' : \mathbf{ty}/\,\alpha\,\{\Gamma\} \quad \beta[x] = \beta'[x] : \mathbf{ty}\,\{\Gamma, x : \alpha\}}{\beta = \beta' : \mathbf{ty}/\,\alpha\,\{\Gamma\}}$$

Non-exclusive introduction forms

$$\text{F:08.05}\mathbf{slabs}\ \frac{\beta : \mathbf{ty}\,\{\Gamma, x : \alpha\}}{\Lambda_x \beta : \mathbf{ty}/\,\alpha\,\{\Gamma\}} \qquad \text{F:12.07}\beta\ \frac{\beta : \mathbf{ty}\,\{\Gamma, x : \alpha\} \quad \gamma : \Gamma\,\{\Delta\} \quad a : \mathbf{el}\,\alpha\gamma\,\{\Delta\}}{((\Lambda_x \beta)\gamma)[a] = \beta(\gamma, x \leftarrow a) : \mathbf{ty}\,\{\Delta\}}$$

dependent function types, application, abstraction

Global premise: $\Gamma : \mathbf{cxt}$

$$\text{F:07.05}\mathbf{form}\ \frac{\alpha : \mathbf{ty}\,\{\Gamma\} \quad \beta : \mathbf{ty}/\,\alpha\,\{\Gamma\}}{\Pi\,\alpha\,\beta : \mathbf{ty}\,\{\Gamma\}}$$

$$\text{F:12.08}\mathbf{form}_c\ \frac{\alpha = \alpha' : \mathbf{ty}\,\{\Gamma\} \quad \beta = \beta' : \mathbf{ty}/\,\alpha\,\{\Gamma\}}{\Pi\,\alpha\,\beta = \Pi\,\alpha'\,\beta' : \mathbf{ty}\,\{\Gamma\}} \qquad \text{F:12.09}\mathbf{form}_s\ \frac{\alpha : \mathbf{ty}\,\{\Gamma\} \quad \beta : \mathbf{ty}/\,\alpha\,\{\Gamma\} \quad \gamma : \Gamma\,\{\Delta\}}{(\Pi\,\alpha\,\beta)\gamma = \Pi\,(\alpha\gamma)\,(\beta\gamma) : \mathbf{ty}\,\{\Delta\}}$$

Further global premises: $\alpha : \mathbf{ty}\,\{\Gamma\}, \quad \beta : \mathbf{ty}/\,\alpha\,\{\Gamma\}$

$$\text{F:05.05}\mathbf{app}\ \frac{f : \mathbf{el}\,(\Pi\,\alpha\,\beta)\,\{\Gamma\} \quad a : \mathbf{el}\,\alpha\,\{\Gamma\}}{f\,@\,a : \mathbf{el}\,\beta[a]\,\{\Gamma\}}$$

$$\text{F:10.07}\mathbf{app}_c\ \frac{f = f' : \mathbf{el}\,(\Pi\,\alpha\,\beta)\,\{\Gamma\} \quad a = a' : \mathbf{el}\,\alpha\,\{\Gamma\}}{f\,@\,a = f'\,@\,a' : \mathbf{el}\,\beta[a]\,\{\Gamma\}} \qquad \text{F:10.08}\mathbf{app}_s\ \frac{f : \mathbf{el}\,(\Pi\,\alpha\,\beta)\,\{\Gamma\} \quad a : \mathbf{el}\,\alpha\,\{\Gamma\} \quad \gamma : \Gamma\,\{\Delta\}}{(f\,@\,a)\gamma = (f\gamma)\,@(a\gamma) : \mathbf{el}\,(\beta[a])\gamma\,\{\Delta\}}$$

Global side-condition: (x Γ-fresh)

$$\text{F:10.10}\zeta\ \frac{f, f' : \mathbf{el}\,\Pi\,\alpha\,\beta\,\{\Gamma\} \quad f\,@\,x = f'\,@\,x : \mathbf{el}\,\beta[x]\,\{\Gamma, x : \alpha\}}{f = f' : \mathbf{el}\,(\Pi\,\alpha\,\beta)\,\{\Gamma\}}$$

Non-exclusive introduction forms

$$\text{F:05.06}\mathbf{abs}\ \frac{b : \mathbf{el}\,\beta[x]\,\{\Gamma, x : \alpha\}}{\lambda_x b : \mathbf{el}\,(\Pi\,\alpha\,\beta)\,\{\Gamma\}} \qquad \text{F:10.09}\beta\ \frac{b : \mathbf{el}\,\beta[x]\,\{\Gamma, x : \alpha\} \quad \gamma : \Gamma\,\{\Delta\} \quad a : \mathbf{el}\,\alpha\gamma\,\{\Delta\}}{((\lambda_x b)\gamma)\,@\,a = b(\gamma, x \leftarrow a) : \mathbf{el}\,(\beta\gamma)[a]\,\{\Delta\}}$$

dependent ordered pair types

Global premise: $\Gamma : \mathbf{cxt}$

$$\mathbf{form}\ \frac{\alpha : \mathbf{ty}\ \{\Gamma\} \quad \beta : \mathbf{ty}/\,\alpha\ \{\Gamma\}}{\Sigma\,\alpha\,\beta : \mathbf{ty}\ \{\Gamma\}}$$

$$\mathbf{form}_c\ \frac{\alpha = \alpha' : \mathbf{ty}\ \{\Gamma\} \quad \beta = \beta' : \mathbf{ty}/\,\alpha\ \{\Gamma\}}{\Sigma\,\alpha\,\beta = \Sigma\,\alpha'\,\beta' : \mathbf{ty}\ \{\Gamma\}} \qquad \mathbf{form}_s\ \frac{\alpha : \mathbf{ty}\ \{\Gamma\} \quad \beta : \mathbf{ty}/\,\alpha\ \{\Gamma\} \quad \gamma : \Gamma\ \{\Delta\}}{(\Sigma\,\alpha\,\beta)\gamma = \Sigma\,(\alpha\gamma)\,(\beta\gamma) : \mathbf{ty}\ \{\Delta\}}$$

Further global premises: $\alpha : \mathbf{ty}\ \{\Gamma\}, \quad \beta : \mathbf{ty}/\,\alpha\ \{\Gamma\}$

$$\mathbf{lproj}\ \frac{p : \mathbf{el}\ \Sigma\,\alpha\,\beta\ \{\Gamma\}}{\pi_0 p : \mathbf{el}\ \alpha\ \{\Gamma\}} \qquad \mathbf{lproj}_c\ \frac{p = p' : \mathbf{el}\ \Sigma\,\alpha\,\beta\ \{\Gamma\}}{\pi_0 p = \pi_0 p' : \mathbf{el}\ \alpha\ \{\Gamma\}} \qquad \mathbf{lproj}_s\ \frac{p : \mathbf{el}\ \Sigma\,\alpha\,\beta\ \{\Gamma\} \quad \gamma : \Gamma\ \{\Delta\}}{(\pi_0 p)\gamma = \pi_0(p\gamma) : \mathbf{el}\ \alpha\gamma\ \{\Delta\}}$$

$$\mathbf{rproj}\ \frac{p : \mathbf{el}\ \Sigma\,\alpha\,\beta\ \{\Gamma\}}{\pi_1 p : \mathbf{el}\ \beta[(\pi_0 p)]\ \{\Gamma\}} \qquad \mathbf{rproj}_c\ \frac{p = p' : \mathbf{el}\ \Sigma\,\alpha\,\beta\ \{\Gamma\}}{\pi_1 p = \pi_1 p' : \mathbf{el}\ \beta[(\pi_0 p)]\ \{\Gamma\}} \qquad \mathbf{rproj}_s\ \frac{p : \mathbf{el}\ \Sigma\,\alpha\,\beta\ \{\Gamma\} \quad \gamma : \Gamma\ \{\Delta\}}{(\pi_1 p)\gamma = \pi_1(p\gamma) : \mathbf{el}\ (\beta\gamma)[(\pi_0(p\gamma))]\ \{\Delta\}}$$

$$\zeta\ \frac{p, q : \mathbf{el}\ (\Sigma\,\alpha\,\beta)\ \{\Gamma\} \quad \pi_0 p = \pi_0 q : \mathbf{el}\ \alpha\ \{\Gamma\} \quad \pi_1 p = \pi_1 q : \mathbf{el}\ \beta[\pi_0 p]\ \{\Gamma\}}{p = q : \mathbf{el}\ (\Sigma\,\alpha\,\beta)\ \{\Gamma\}}$$

Non-exclusive introduction forms

$$\mathbf{pair}\ \frac{a : \mathbf{el}\ \alpha\ \{\Gamma\} \quad b : \mathbf{el}\ \beta[a]\ \{\Gamma\}}{\langle a, b\rangle : \mathbf{el}\ (\Sigma\,\alpha\,\beta)\ \{\Gamma\}} \qquad \beta_0\ \frac{a : \mathbf{el}\ \alpha\ \{\Gamma\} \quad b : \mathbf{el}\ \beta[a]\ \{\Gamma\}}{\pi_0\langle a, b\rangle = a : \mathbf{el}\ \alpha\ \{\Gamma\}} \qquad \beta_1\ \frac{a : \mathbf{el}\ \alpha\ \{\Gamma\} \quad b : \mathbf{el}\ \beta[a]\ \{\Gamma\}}{\pi_1\langle a, b\rangle = b : \mathbf{el}\ \beta[a]\ \{\Gamma\}}$$

type o of sets, type-family ι of elements of a set

$$\frac{\Gamma : \mathbf{cxt}}{o : \mathbf{ty}\,\{\Gamma\}} \qquad \frac{\gamma : \Gamma\,\{\Delta\}}{o\gamma = o : \mathbf{ty}\,\{\Delta\}} \qquad \frac{\Gamma : \mathbf{cxt}}{\iota : \mathbf{ty}/\,o\,\{\Gamma\}} \qquad \frac{\gamma : \Gamma\,\{\Delta\}}{\iota\gamma = \iota : \mathbf{ty}/\,o\,\{\Delta\}}$$

empty type?

In addition to the above (for a generic ground family):

$$\frac{a, b : \mathbf{el}\ o_{\mathbb{O}}\ \{\Gamma\}}{b = a : \mathbf{el}\ o_{\mathbb{O}}\ \{\Gamma\}} \qquad \frac{a, b : \mathbf{ty}/\,o_{\mathbb{O}}\ \{\Gamma\}}{b = a : \mathbf{ty}/\,o_{\mathbb{O}}\ \{\Gamma\}}$$

Remarks:

The empty type $o_{\mathbb{O}}$ comes with its own family of types $\iota_{\mathbb{O}} : \mathbf{ty}/\,o_{\mathbb{O}}$. This is how we might represent/host the completely empty type-theory, with no types.

There really isn't anything here to say that $o_{\mathbb{O}}$ is initial. That would say, among other things, that the function type $\Pi\, o_{\mathbb{O}}\, \iota_{\mathbb{O}}$ has an inhabitant: 'the' empty function.

singleton type

Beside generic stuff:

$$\mathbf{sing}\ \frac{\Gamma : \mathbf{cxt}}{0 : \mathbf{el}\ o_{\mathbb{1}}\ \{\Gamma\}} \qquad \mathbf{sing}^{s}\ \frac{\gamma : \Gamma\,\{\Delta\}}{0 = 0\gamma : \mathbf{el}\ o_{\mathbb{1}}\ \{\Delta\}}$$

$$\frac{a, b : \mathbf{el}\ o_{\mathbb{1}}\ \{\Gamma\}}{b = a : \mathbf{el}\ o_{\mathbb{1}}\ \{\Gamma\}} \qquad \frac{\alpha : \mathbf{ty}\,\{\Gamma\}}{\mathbf{K}\,\alpha : \mathbf{ty}/\,o_{\mathbb{1}}\ \{\Gamma\}} \qquad \frac{\alpha : \mathbf{ty}\,\{\Gamma\} \quad a : \mathbf{el}\ o_{\mathbb{1}}\ \{\Gamma\}}{(\mathbf{K}\alpha)[a] = \alpha : \mathbf{ty}\,\{\Gamma\}}$$

Remarks:

The standard singleton type $o_{\mathbb{1}}$ comes with for each type α a family of types indexed over $o_{\mathbb{1}}$, namely the family with constant value α. (As a type-theory, this has a single type, namely α.) Might postulate $\iota_{\mathbb{1}} = \mathbf{K}(o_{\mathbb{O}}) : \mathbf{ty}/\,o_{\mathbb{1}}$.

Bureacratic desiderata

To flush errors out of the rules, and maybe even to simplify them, one can consider requiring that certain rules should be *admissible*. In no settled order:

(i) If J is any judgement, and $J\,\{\Gamma\}$ is provable, then $\Gamma : \mathbf{cxt}$ should be provable.

The same goes if any of

$$\gamma : \Delta\,\{\Gamma\}, \quad \gamma = \gamma' : \Delta\,\{\Gamma\},$$
$$\delta : \Gamma\,\{\Delta\}, \quad \delta = \delta' : \Gamma\,\{\Delta\}$$

are provable.

(ii) If either of $a : \mathbf{el}\ \alpha\,\{\Gamma\}$ or $b : \mathbf{ty}/\,\alpha\,\{\Gamma\}$ are provable, then $\alpha : \mathbf{ty}\,\{\Gamma\}$ should be provable.

(iii) If either of $a = a' : \mathbf{el}\ \alpha\,\{\Gamma\}$ $b = b' : \mathbf{ty}/\,\alpha\,\{\Gamma\}$ are provable, then not only $\alpha : \mathbf{ty}\,\{\Gamma\}$, but also $a, a' : \mathbf{el}\ \alpha\,\{\Gamma\}$ (or $b, b' : \mathbf{ty}/\,\alpha\,\{\Gamma\}$ as appropriate) should be provable.

(iv) If $\alpha = \alpha' : \mathbf{ty}\,\{\Gamma\}$ is provable, then $\alpha, \alpha' : \mathbf{ty}\,\{\Gamma\}$ should be provable.

(v) If $\gamma = \gamma' : \Gamma\,\{\Delta\}$ is provable, then $\gamma, \gamma' : \Gamma\,\{\Delta\}$ should be provable.

How to check and organise this. One wants a robust notion of rule/production, yet without the horrible prolixity evident in these pages.

I have already silently used a convention of 'commas' to avoid painful repetition.

As far as I can see, there are only two rules by which to infer that something is a context; and only two rules to infer that something is a subcontext of something else.

Should it be that if $\gamma : ()\,\{\Gamma\}$ then $\Gamma : \mathbf{cxt}$?

attempt at path types

Probably I have misunderstood $\mathbb{I}$, and declarations $i : \mathbb{I}$.

$$\mathbb{I}\textbf{form}\ \frac{\Gamma : \mathbf{cxt}}{\mathbb{I} : \mathbf{ty}\ \{\Gamma\}} \qquad \mathbb{I}\textbf{in}_{0,1}\ \frac{\Gamma : \mathbf{cxt}}{0, 1 : \mathbf{el}\ \mathbb{I}} \qquad \mathbb{I}\textbf{in}_{\vee,\wedge}\ \frac{\phi, \psi : \mathbf{el}\ \mathbb{I}}{\phi \vee \psi, \phi \wedge \psi : \mathbf{el}\ \mathbb{I}} \qquad \mathbb{I}\textbf{in}_{-}\ \frac{\phi : \mathbf{el}\ \mathbb{I}}{\overline{\phi} : \mathbf{el}}$$

Various equations at type $\mathbb{I}$: distributive lattice, extremities, and an involution that exchanges $(0, \vee)$ with $(1, \wedge)$.

The 'elimination' rules for path types are unusual. In some respects, a path type is like an exponential $\_^{\mathbb{I}}$. In other respects, it vaguely resembles a pair type, with projections (at the extremities).

Global premises: $\alpha : \mathbf{ty}\ \{\Gamma\}$

$$\textbf{form}\ \frac{a, b : \mathbf{el}\ \alpha\ \{\Gamma\}}{\mathbf{P}_\alpha\ a\ b : \mathbf{ty}\ \{\Gamma\}}$$

$$\textbf{elimination}\ \frac{a, b : \mathbf{el}\ \alpha\ \{\Gamma\} \quad p : \mathbf{el}\ \mathbf{P}_\alpha\ a\ b\ \{\Gamma\} \quad r : \mathbf{el}\ \mathbb{I}\ \{\Gamma\}}{p\langle r\rangle : \mathbf{el}\ \alpha\ \{\Gamma\}}$$

$$\textbf{extremity}_0\ \frac{a, b : \mathbf{el}\ \alpha\ \{\Gamma\} \quad p : \mathbf{el}\ \mathbf{P}_\alpha\ a\ b\ \{\Gamma\}}{p\langle 0\rangle = a : \mathbf{el}\ \alpha\ \{\Gamma\}}$$

$$\textbf{extremity}_1\ \frac{a, b : \mathbf{el}\ \alpha\ \{\Gamma\} \quad p : \mathbf{el}\ \mathbf{P}_\alpha\ a\ b\ \{\Gamma\}}{p\langle 1\rangle = b : \mathbf{el}\ \alpha\ \{\Gamma\}}$$

$$\zeta\ \frac{a, b : \mathbf{el}\ \alpha\ \{\Gamma\} \quad p, p' : \mathbf{el}\ \mathbf{P}_\alpha\ a\ b \quad p\langle i\rangle = p'\langle i\rangle : \mathbf{el}\ \alpha\ \{\Gamma, i : \mathbb{I}\}}{p = p' : \mathbf{el}\ \mathbf{P}_\alpha\ a\ b\ \{\Gamma\}}$$

$$\textbf{introduction}\ \frac{a : \mathbf{el}\ \alpha\ \{\Gamma, i : \mathbb{I}\}}{\langle i\rangle.a : \mathbf{el}\ \mathbf{P}_\alpha\ (a(i \leftarrow 0))\ (a(i \leftarrow 1))\ \{\Gamma\}}$$

$$\textbf{introduction-alt}\ \frac{t : \mathbf{el}\ \alpha\ \{\Gamma, i : \mathbb{I}\} \quad a = t(i \leftarrow 0) : \mathbf{el}\ \alpha\ \{\Gamma\} \quad b = t(i \leftarrow 1) : \mathbf{el}\ \alpha\ \{\Gamma\}}{\langle i\rangle.t : \mathbf{el}\ \mathbf{P}_\alpha\ a\ b\ \{\Gamma\}}$$

$$\beta\ \frac{t : \mathbf{el}\ \alpha\ \{\Gamma, i : \mathbb{I}\} \quad \gamma : \Delta\ \{\Gamma\} \quad r : \mathbf{el}\ \mathbb{I}\ \{\Delta\}}{((\langle i\rangle.t)\gamma)\langle r\rangle = t(\gamma, i \leftarrow r) : \mathbf{el}\ \alpha\gamma\ \{\Delta\}}$$

There is something awkward and puzzling here, connected with substitutions: for variables associated with $\mathbb{I}$ (dimensions?), versus those associated with the ordinary kind of type. Declarations $i : \mathbb{I}$ seem special somehow, as if they can always migrate to the front (or rear) of a context. ?? Should they constitute a special kind of 'dimensional' context? Should there be special substitutions for dimensions, like $(i0)$ in cubical TT?

I may not be making enough use of the notion of a family. How is the apparatus for transporting between elements of types in a family supposed to arise? In a way, that is the *real* use we make of an element of a path type.

What if one were to try to express a type of paths between two given *types*? (This is something Conor McBride has long wanted, inexplicably; it would be something additional to 'ordinary' path types between elements of a universe, should those types both be in a universe.)

It seems to me that there may be a danger of inconsistency with such 'type path' types, as one might be able to emulate a type of all types. Or, just imaginably, something even more surprising may show up, like classical logic.

remarks on self-description and -hosting

What is a dependent type theory? An answer should avoid circularity. We might use a DTT we already know as a 'host', and explain how to represent arbitrary 'guest' type-theories. If you already know perfectly well what a dependent type theory is, and understand one such as the one which closes a ground family of types under operators such as Π and Σ, then I can show you what a dependent type theory looks like when its types are made the indices of the ground type, and it is 'hosted' in the one you understand (whatever precisely that means). Some insight might arise in this way, although some puzzlement too.

It might seem as if we have to know already what a DTT is, and indeed a specific one, in order to be told the answer.

Isn't it exasperating when an evangelist pitches us a doctrine we can't even begin to understand, and then tells us (when we ask what we'd be affirming, ie. what it means), that if only we believe, we will understand what it is we believe?

What can we say? There are various forms of judgments, categorical (outright), and hypothetical (qualified, or context-sensitive).

One of the categorical forms says that such and such a thing is a type, and another that such and such another thing inhabits such a type. There may be other judgement forms in which types appear, such as equations between types or elements of types. Indeed, in the type theory in previous pages there are judgement forms pertaining to families-of/slices-over types, depending on or indexed over given elements of such and such a type. At any rate, in a type-theory, as the name suggests, the judgments concern types, their inhabitants, and other pertinent entities.

The hypotheses of a hypothetical judgement form a context, which in our type theory is an ascription of types to variable names.

Then there are formal rules for establishing judgments, and so on. What is being presupposed here is the very idea of a logical or inferential system, centred around the notion of judgment. A categorical judgment is something can stands at the conclusion of an inference step, established by it. A hypothetical judgment is something that can stand among the premises, and be put to work in justifying of the step.

A similar kind of circularity occurs in connection with virtual machines. A hypervisor is a program that may be run under a common operating system, that provides environments that look exactly like 'bare machines' to operating-system code. When it is running, a hypervisor can 'host' several virtual machines, including another instance of the operating system it runs under itself. The fundamental concept is 'machine', which we cannot explain in terms of the notion 'virtual machine'.

Let us look into what it means for a type-theory to 'host itself'. To begin with, we had better simplify the foregoing logical framework drastically. Some proposals might be:

(i) throwing out named variables (replacing them in effect by pointers to positions in the context – at least in theory),

(ii) throwing out subcontexts,

(iii) throwing out families/slices/type-valued functions (dealing with them as functions),

(iv) throwing out tuples and any such thing.

(v) throwing out substitutions. This is rather drastic, but I shall try to work them back in.

We are left with ground types o and the small types $\iota\, a$ indexed by $a : o$, and closure of the types under dependent products (written with bound variables – or perhaps combinators). The signature (π, @, λ) and equations (β, η) for dependent products can perhaps be written as follows. (I use '**id**' for the identity function, and $\mathbf{e}\,\alpha$ for the endo-type $\alpha \to \alpha$. I use $(\cdot)$ for composition, and the infix symbol @ for application.

In any context,

$$\pi \quad : (a : o) \to (\iota\, a \to o) \to o$$

Any context extending $a : o, b : \iota\, a \to o$ can itself be extended by two typed constants, with two typed equations, abbreviated to $\iota(\pi\, a\, b) \cong (x : \iota\, a) \to \iota(b\, x)$

$$\begin{array}{lcl} @ & : & \iota(\pi\, a\, b) \to (x : \iota\, a) \to \iota(b\, x) \\ \lambda & : & ((x : \iota\, a) \to \iota(b\, x)) \to \iota(\pi\, a\, b) \\ \beta & : & @ \cdot \lambda = \mathbf{id} \ : \mathbf{e}\,((x : \iota\, a) \to \iota(b\, x)) \\ \eta & : & \lambda \cdot @ = \mathbf{id} \ : \mathbf{e}\,(\iota(\pi\, a\, b)) \end{array}$$

Let's also write it without bound variables variables in the types: (so $a \to b$ is $\Pi\, a\, (K\, b)$)

$$\begin{array}{lcl} @ & : & \iota(\pi\, a\, b) \to \Pi(\iota\, a)(\iota \cdot b) \\ \lambda & : & \Pi(\iota\, a)(\iota \cdot b) \to \iota(\pi\, a\, b) \\ \beta & : & @ \cdot \lambda = \mathbf{id} \ : \mathbf{e}\,(\Pi(\iota\, a)(\iota \cdot b)) \\ \eta & : & \lambda \cdot @ = \mathbf{id} \ : \mathbf{e}\,(\iota(\pi\, a\, b)) \end{array}$$

But we should also remember: in any context:

$$\pi : \Pi\, o\, ((\to o) \cdot (\to o) \cdot \iota)$$

This is in a sense a self-description. We use an extensive type theory (with dependent function types and a universe) on the meta-level to host a description of (the function part of) itself at object-level. The LF's description of itself more or less identifies the function space of the LF with that of the hosted theory. It says there is an isomorphism between $(x : \iota\,\alpha) \to \iota(\beta\,x)$ and $\iota(\pi\,\alpha\,\beta)$. The components of the isomorphism are λ and @.

How might things work if we were to take account of substitutions? Our simple β equations become more difficult. If I introduce an explicit operator $a \otimes \gamma$ for the application of a substitution to an expression, (and continue to write substitution extension with an infix comma, some relevant equations are:

$$
\begin{aligned}
& ((\lambda\,b) \otimes \gamma)@a = b \otimes (\gamma, a) \\
= \\
& ((\lambda\,b) \otimes \gamma)@ = (b\otimes) \cdot (\gamma,) \\
= \\
& (((\lambda\,b)\otimes)\gamma)@ = (b\otimes) \cdot ((,)\,\gamma) \\
= \\
& @ \cdot ((\lambda\,b)\otimes) = ((b\otimes)\cdot) \cdot (,) \\
= \\
& ((@\cdot) \cdot \otimes \cdot \lambda)\,b = ((b\otimes)\cdot) \cdot (,) \\
= \\
& (@\cdot) \cdot \otimes \cdot \lambda = \text{something involving } \otimes \text{ and } (,) \text{ in lots of dots}
\end{aligned}
$$

Perhaps $(\cdot(,)) \cdot (\cdot) \cdot \otimes)$. I randomly chose to eliminate γ before b. I have yet to look at the types.

The self-description uses dependent function types (expressed either in bound-variable notation, or by using linear combinators such as composition) on the meta-level, as well as application (expressed $f\,a$). It does not use abstraction much, except to define composition and the identity.

The (simple) equations each have the form that a certain function (expressed using meta-level composition, defined by meta-level linear abstraction) is/equals the identity. In the case of β that is the identity on a meta-level function type of level 1, meaning with ι-formed domain and codomain. In the case of η that is the identity on a meta-level ground type.

It may be worth noting that there are so far no equations between functions with codomain o. (We encounter some in a moment.)

In a certain sense the signature and equations above reflect the host's functional type structure inside (o, ι) . But we may try to reflect the host's *universe* type structure inside o.

$$
\begin{array}{lcll}
\hat{o} & : & o & \\
\hat{\iota} & : & \iota\,\hat{o} \to o & \\
 & & & \\
\hat{\pi} & : & (a : \iota\,\hat{o}) \to (\iota(\hat{\iota}\,a) \to \iota\,\hat{o}) \to \iota\,\hat{o} & \\
\hat{\beta} & : & \hat{\iota}(\hat{\pi}\,a\,b) = \pi\,(\hat{\iota}\,a)(\hat{\iota}\cdot b) : o & [\, a : \iota\,\hat{o}, b : \iota(\hat{\iota}\,a) \to \iota\,\hat{o}\,] \\
 & & & \\
\hat{\hat{o}} & : & \iota\,\hat{o} & \\
\hat{\hat{\iota}} & : & \iota(\hat{\iota}\,\hat{\hat{o}}) \to \iota\,\hat{o} & \\
 & & & \\
\hat{\hat{\pi}} & : & (a : \iota(\hat{\iota}\,\hat{\hat{o}})) \to (\iota(\hat{\iota}\,(\hat{\hat{\iota}}\,a)) \to \iota\,(\hat{\iota}\,\hat{\hat{o}})) \to \iota\,(\hat{\iota}\,\hat{\hat{o}}) & \\
\hat{\hat{\beta}} & : & \hat{\hat{\iota}}(\hat{\hat{\pi}}\,a\,b) = \hat{\pi}\,(\hat{\hat{\iota}}\,a)(\hat{\hat{\iota}}\cdot b) : \iota\,\hat{o} & [\, a : \iota(\hat{\iota}\,\hat{\hat{o}}), b : \iota(\hat{\iota}(\hat{\hat{\iota}}\,a)) \to \iota(\hat{\iota}\,\hat{\hat{o}})\,] \\
 & : & &
\end{array}
$$

To self-host, the levels of function types in the host type theory need be no higher than 2. This means that binding operators (such as λ) just bind variables of level 0. This kind of binding is comparatively simple. We can 'bootstrap' function types of arbitrary levels. [What exactly does this mean?]

Is the equational theory of π, @ and λ decidable? This seems an interesting question. Can we dispense with any notation for proofs of equations? (In that case, equational reasoning can be 'inferred'.)

What if definitional equations in a hosted type theory were expressed as "identity types" in the logical framework, using doubly indexed families? This means that we are dealing not only with o, ι but with other types $a =_o b$, $c =_{\iota a} d$, $(\cdots) =_{(\cdots)} (\cdots)$. Equations would then have real proofs, as would rules of equational reasoning. Perhaps though we need not carry these proofs about, or care about them, if the equational theory has good properties.

Σ

$$\textstyle\sum \;\; : (a : o) \to (\iota\, a \to o) \to o$$

In a context $a : o, b : \iota\, a \to o$,

$$\begin{array}{rcl}
\mathbf{p}_0 & : & \iota(\textstyle\sum\, a\, b) \to \iota\, a \\
\mathbf{p}_1 & : & (z : \iota(\textstyle\sum\, a\, b)) \to \iota\, (b\, (\mathbf{p}_0\, z)) \\
\mathbf{pr} & : & ((x : \iota\, a) \to \iota(b\, x) \to \iota(\textstyle\sum\, a\, b) \\
\beta_0\, (x : \iota\, a) & : & \mathbf{p}_0 \cdot \mathbf{pr}\, x = \mathbf{const}\, x : \iota\, (b\, x) \to \iota\, a \\
\beta_1\, (x : \iota\, a) & : & \mathbf{p}_1 \cdot \mathbf{pr}\, x = \mathbf{id} : \mathbf{e}\, (\iota(b\, x)) \\
\eta & : & (z \mapsto \mathbf{pr}(\mathbf{p}_0\, z)(\mathbf{p}_1\, z)) = \mathbf{id} : \mathbf{e}\, (\iota(\textstyle\sum\, a\, b))
\end{array}$$

If one assumes a 'native' type $((x : \alpha) \times \beta\, x)$ of ordered pairs, with projectors π_0 and π_1, and pairing $\langle a, b\rangle$:

$$\begin{array}{rcll}
(z \mapsto \langle \mathbf{p}_0\, z, \mathbf{p}_1 z\rangle) & : & \iota(\textstyle\sum\, a\, b) \to ((x : \iota\, a) \times \iota(b\, x)) & \\
\mathbf{pr} & : & (((x : \iota\, a) \times \iota(b\, x)) \to \iota(\textstyle\sum\, a\, b) & \\
\beta & : & (z \mapsto \langle \mathbf{p}_0\, z, \mathbf{p}_1 z\rangle) \cdot \mathbf{pr} = \mathbf{id} & : \mathbf{e}\, ((x : \iota a) \times \iota(b\, x)) \\
\eta & : & \mathbf{pr} \cdot (z \mapsto \langle \mathbf{p}_0\, z, \mathbf{p}_1 z\rangle) = \mathbf{id} & : \mathbf{e}\, (\iota(\textstyle\sum\, a\, b))
\end{array}$$

2

In the same way as one has o and ι, maybe one could have:

$$\begin{array}{rll}
o_0 = o_{\mathbb{O}} : \mathbf{ty} & - & \iota_0 : \mathbf{ty}/\, o_{\mathbb{O}} \\
o_1 = o_{\mathbb{1}} : \mathbf{ty} & 0_1 : \mathbf{el}\, o_{\mathbb{1}} & \iota_1 : \mathbf{ty}/\, o_{\mathbb{1}} \\
\mathbb{2} : \mathbf{ty} & 0_2, 1_2 : \mathbf{el}\, \mathbb{2} & \iota_2 : \mathbf{ty}/\, \mathbb{2}
\end{array}$$

with

$$\begin{array}{ll}
\iota_1[0_1] = o_{\mathbb{O}} : \mathbf{ty}, & \iota_2[0_2] = o_{\mathbb{O}} : \mathbf{ty} \\
& \iota_2[1_2] = o_{\mathbb{1}} : \mathbf{ty}
\end{array}$$

- o_0 the completely empty type theory $\{\}$.
- o_1 the singleton type theory $\{\{\}\}$, with one type, which is empty.
- o_2 the boolean type theory $\{\{\}, \{\{\}\}\}$, with two types, which are empty and singleton.

One needs to be super-careful. A singleton may be OK, and (just conceivably) too an empty set, but not anything like $(+1)$, which introduces choices and alternatives [4]. It is important not to compromise or complicate what one can guarantee about extensionality – inferring equational judgments with variables as left or right terms.

remarks on type-families

No doubt there are many mistakes above. From time to time I notice some, ask myself what I was thinking, and investigate further. Some areas are murkier than others.

Perhaps the first cloud of questions forms itself around the judgement form $f : \mathbf{ty}/\, \alpha$, that is without precise counterpart in conventional presentations of dependent type theory. Do these 'conventional presentations' skirt rather awkwardly about the matter? In conventional presentations, one has just the familiar judgement forms $a : \mathbf{el}\, \alpha$ and $a = a' : \mathbf{el}\, \alpha$, and explaining what type α is amounts to explaining what one can do with objects/elements of type α (or their 'role' in communicative life) and (as Martin-Löf would have it) when such objects are equal, or the same object of that type. Does one now have also to explain how one can form slices/families over α, and their equality?

No doubt, there are important and subtle things that might be made clearer by using better chosen words. But the issue here is the rôle of the judgement form $\ldots : \mathbf{ty}/\, \alpha$ (about families over a type), that looks as if it stands at the same level (in some sense) as the $\ldots : \mathbf{el}\, \alpha$ (about objects of a type). Besides these one has the equational judgements $\ldots = \ldots : \mathbf{ty}/\, \alpha$ and $\ldots = \ldots : \mathbf{el}\, \alpha$. But the basic question pertains not so much to the (binary) equational forms of judgement (criteria for equality) versus unary forms (criteria for application), but rather why one needs to have the very concept of a slice/family/type-valued 'function', that may be 'the same', whatever the exactly terminology one uses.

What follows are a few thoughts I have entertained over many years connected with this issue. They are (for the time being) in a disorganised, crude, and tentative form.

Very crudely, a thing 'is' what it 'does', or to put it in a slightly better way, what *we* do with it. Forgetting (for the moment) about type-families, one might say, again very crudely, that what we do with a type is to classify – or certify – things (objects of that type, families over that type), and to equate them (saying essentially that those things are equal because they serve their purpose equally well).

To explain a type α, say what can be done with objects of type α – what they are good for, the value of access to such a thing, what you can get out of it. Why you might be sad if you deprived of it.

[4] Except when applied to $o_{\mathbb{O}}$!

Something we can do with an object is to 'say' something about it, or perhaps more generally use it as an index to pick one of a family/array of types, or as an argument to determine the 'value' of a type-valued function. In the case of 'saying', we have an array of propositions. In the more general case, an array of types. At any rate, one role of an object is to index families/arrays. To be is to be the value of a bound (index) variable.

What gets done with an object a of type o? It is used to identify a type $\iota\, a$. It indexes that family. However, $\iota\, a$ is actually not a type, but a mark of a judgement form.

There is at least one family that is not given by abstracting a variable, namely the ground family (o, ι). This is in a sense the 'generic' dependent family. (It is a place holder for a hosted type theory.)

The function of the machinery of families. We have variable binding, and instantiation, ie 'functions', but only in a very simple context: the codomain of our 'function' is always 'type', and this is not a domain type. Reminiscent: a calculus of classes. Then we have full-blooded, strictly-so-called function types, with an honest-to-goodness codomain that is given by a family of types.

remarks on contexts, substitutions

A (pre-?)context Δ is a finite map from names to types. It has a domain $\overline{\Delta}$ (the finite set of names at which it is defined), and for each name x in the domain, a type $\Delta[x]$[5] responding to that name. Names are inexhaustible, and have a mechanically decidable equality. It is decidable whether a name is associated with a type-expression in the domain of a context (or is fresh to it), and the looking up the type-expression assigned to a name is mechanically computable.

One point that occurs to me about the preorder of contexts, is that it is defined using literal equality between type declarations, rather than judgemental type equality.

$$\frac{\Gamma \sqsubseteq \Delta \quad \alpha : \mathbf{ty}\,\{\Gamma\}}{(\Gamma, x : \alpha) \sqsubseteq \Delta}\ (x\ \Gamma\text{-fresh})\ (x : \alpha \text{ in } \Delta)$$

It might instead look something like this.

$$\frac{\Gamma \sqsubseteq \Delta \quad \alpha, \alpha' : \mathbf{ty}\,\{\Gamma\} \quad \alpha = \alpha' : \mathbf{ty}\,\{\Delta\}}{(\Gamma, x : \alpha) \sqsubseteq \Delta}\ (x\ \Gamma\text{-fresh})\ (\Delta[x] = \alpha')$$

The new side-condition says that Δ has x in its domain, and its value there (indicated by 'array' brackets) is to be known in the rule as α'.

Or this.

$$\frac{\Gamma \sqsubseteq \Delta \qquad \alpha = \Delta[x] : \mathbf{ty}\,\{\Delta\}}{(\Gamma, x : \alpha) \sqsubseteq \Delta}\ (x\ \Gamma\text{-fresh})$$

A substitution δ is also a map on names, with a domain $\overline{\delta}$ that is a decidable set of names x. The value associated with a name x is now $\delta[x]$: a term or object of a type. (Could it be a type-former?) If it is well formed, it will have a domain/input context, and a codomain/output context.

A substitution implements its codomain context in terms of its domain context. This means that anything based on the codomain can be pulled back along the substitution to be based on the domain. 'Anything' means types, terms of types, types indexed over a given type, equations between these things.

In a context, we declare variables for values of certain types. What would go wrong if we allowed also the declaration of variables for families/slices? (This is reminiscent of the δ-declarations in a Dybjer-Setzer IR-code.)

On more bureaucratic matters, Erik Palmgren has (in an exposition of logic with dependent sorts) some suggestive ideas.

- There is a precise notion of 'presupposition' that may be useful. Some judgements – such as those of the form $\Gamma : \mathbf{cxt}$ – can be proved by only one rule (depending on the shape of Γ): when such a judgement occurs as a premise somewhere, the premises of that rule are 'presupposed'. This may help to cut down on the great clutter of premises in most rules.

- He thinks of substitution judgements as actually sets (or sequences, 'conjunctions', or other structures) of 'has type' judgements. (Why does he not regard 'context' judgements as conjunctions of 'is a type' judgements.) This may save multiplying the number of judgement forms.

I cannot remember how the notion of a family emerges in Erik's exposition. There is certainly no explicit judgement form devoted to it.

Perhaps there is another terminology than 'family', 'dependent family of types', slice, fibre-over, and so on: namely, the programmers terminology of array. (See the discussion of static typing by Hoare around 1968.) Certainly arrays are indexed. Usually, there is a type of indices, usually finite in some sense, and the arrays are homogeneous, meaning that all entries share the same type. But one can imagine a special kind of pseudo-array that 'stores' (homogeneously) types. Such an array might be used as a 'data dictionary', recording a heterogeneous type for a more hum-drum array (that actually stores values, though the type of a value can be index dependent).

[5] Nothing to do with families (I presume)?