CHAPTER 1

Computational Type Theories

As alluded to at the end of the previous chapter, we may add a judgement $M \in A$ which deals directly with the objects M which verify the propositions (or types) P. We will develop a full theory of dependent types in the sense of Martin-Löf.[1]

1. The judgments of a computational type theory

Because we will need to consider the introduction of types which do not have a trivial (intensional) equality relation, we must first amend the meaning explanations for some of our judgements, and add a few new forms of judgement. First, we will refer to *types* rather than *propositions* in order to emphasize the generality of the theory; in some presentations, the word *set* is used instead.

The meaning of hypothetical and general judgement are the same as in the previous chapter, and so we will not reproduce them here. The first form of judgement is $\boxed{A \ type}$, and its meaning explanation is as follows:

> To know A *type* is to know an A' such that $A \Rightarrow A'$, and you know what counts as a canonical verification of A' and when two such verifications are equal.

The next form of judgement is $\boxed{M \in A}$, which remains the same as before:

> To know $M \in A$ (presupposing A *type*) is to know an M' such that $M \Rightarrow M'$ and M' is a canonical verification of A.

We'll need to add new judgements for equality (equality of verifications, and equality of types respectively). First, equality of verifications is written $\boxed{M = N \in A}$, and means the following:

> To know $M = N \in A$ (presupposing $M \in A$ and $N \in A$, and thence A *type*) is to know that the values of M and N are equal as canonical verifications of A.

[1]Please note that the judgements given here, and their meaning explanations, are *not* the same as those used in Constable et al's "Computational Type Theory" and Nuprl. In this chapter, we use the term "computational type theory" in a general sense to characterize a family of type theories which have their origin in Martin-Löf's 1979 paper *Constructive Mathematics and Computer Programming*.

The fact that M and N reduce to canonical values which verify A is known from the presuppositions of the judgement; and what it means for them to be equal as such is known from the evidence of the presupposition A *type* which is obtained from the other presuppositions.

For equality of types $\boxed{A = B\ \mathit{type}}$, there are a number of possible meaning explanations, but we'll use the one that Martin-Löf used starting in 1979:

> To know $A = B$ *type* (presupposing A *type* and B *type*) is to know $|_M\, M \in B$ $(M \in A)$ and $|_M\, M \in A$ $(M \in B)$, and moreover $|_{M,N}\, M = N \in B$ $(M = N \in A)$ and $|_{M,N}\, M = N \in A$ $(M = N \in A)$.

In other words, two types are equal when they have the same canonical verifications, and moreover, the same equality relation over their canonical verifications. Note that there are other possible explanations for type equality, including more intensional ones that appeal to the syntactic structure of type expressions, and these turn out to be more useful in proof assistants for practical reasons. However, the extensional equality that we have expounded is the easiest and most obvious one to formulate, so we will use it here.

Now, because we are allowing the definition of types with arbitrary equivalence relations, we cannot use plain hypothetico-general judgement in the course of defining our types. For instance, if we were going to try and define the function type $A \supset B$ in the same way as we did in the previous chapter, we would permit "functions" which are not in fact functional, i.e. they do not take equal inputs to equal outputs. As such, we will need to bake functionality (also called extensionality) into the definition of functions, and since we will need this in many other places, we elect to simplify our definitions by baking it into a single judgement which is meant to be used instead of plain hypothetico-general judgement.

The judgement which expresses simultaneously generality, hypothesis and functionality has been written in multiple ways. Martin-Löf has always written it as $\mathcal{J}\ (\Gamma)$, but this is a confusing notation because it appears as though it is merely a hypothetical judgement (but it is much more, as will be seen). Very frequently, it is written with a turnstile, $\Gamma \vdash \mathcal{J}$, and in the early literature surrounding Constable's Computational Type Theory and Nuprl, it was written $\boxed{\Gamma \gg \mathcal{J}}$; we choose this last option to avoid confusion with a similar judgement form which appears in proof-theoretic, intensional type theories; we'll call the judgement form a "sequent".

First, we will add the syntax of contexts into the computation system:

(Canonical)
$$\cdot \Rightarrow \cdot$$
$$(\Gamma, x \in A) \Rightarrow (\Gamma, x \in A)$$

The sequent judgements will be defined simultaneously with two other judgements, $\boxed{\Gamma\ ctx}$ ("Γ is a context") and $\boxed{x \# \Gamma}$ ("x is fresh in Γ"). Their meaning explanations are as follows:

> To know $\Gamma\ ctx$ is to know that $\Gamma \Rightarrow \cdot$, or it is to know a variable x and expressions Δ, A such that $\Gamma \Rightarrow \Delta, x \in A$ and $\Delta \gg A\ type$, and $x \# \Delta$.
>
> To know $x \# \Gamma$ is to know that $\Gamma \Rightarrow \cdot$, or, if $\Gamma \Rightarrow \Delta, y \in A$ such that x is not y and $x \# \Delta$.

In other words, the well-formed contexts are inductively generated by the following grammar:

$$\frac{}{\cdot\ ctx} \qquad \frac{\Gamma\ ctx \quad \Gamma \gg A\ type \quad x \# \Gamma}{\Gamma, x \in A\ ctx}$$

$$\frac{}{x \# \cdot} \qquad \frac{x \# \Gamma}{x \# \Gamma, y \in A}$$

We will say that $\Gamma \gg \mathcal{J}$ is only a judgement under the presuppositions that $\Gamma\ ctx$ and that $\mathcal{J}$ is a categorical judgement of the form $A\ type$ or $M \in A$. Its meaning explanation must be given separately for each kind of conclusion.

Now we may begin giving the meaning explanations for $\Gamma \gg \mathcal{J}$, starting with typehood with respect to a context:

> To know $\Gamma \gg A\ type$ is to know, if $\Gamma \Rightarrow \cdot$, then $A\ type$; otherwise, if $\Gamma \Rightarrow \Delta, x \in B$, that $|_M\, \Delta \gg [M/x]A\ type$ $(\Delta \gg M \in B)$ and moreover $|_{M,N}\, \Delta \gg [M/x]A = [N/x]A\ type$ $(\Delta \gg M = N \in B)$.

In this way, we have defined the sequent judgement by structural recursion on the context. We can explain type equality sequents in a similar way:

> To know $\Gamma \gg A = B\ type$ is to know, if $\Gamma \Rightarrow \cdot$, that $A = B\ type$; otherwise, if $\Gamma \Rightarrow \Delta, x \in C$, it is to know
>
> $$|_M\, \Delta \gg [M/x]A = [M/x]B\ type\ (\Delta \gg M \in C)$$
>
> and moreover, to know
>
> $$|_{M,N}\, \Delta \gg [M/x]A = [N/x]B\ type\ (\Delta \gg M = N \in C)$$

Next, the meaning of membership sequents is explained:

> To know $\Gamma \gg M \in A$ is to know, if $\Gamma \Rightarrow \cdot$, that $M \in A$; otherwise, if $\Gamma \Rightarrow \Delta, x \in B$, it is to know
>
> $$|_N\, \Delta \gg [N/x]M \in [N/x]A\ (\Delta \gg N \in B)$$

and moreover, to know

$$|_{L,N}\, \Delta \gg [L/x]M = [N/x]M \in A\ (\Delta \gg L = N \in B)$$

Finally, member equality sequents have an analogous explanation:

To know $\Gamma \gg M = M' \in A$ is to know, if $\Gamma \Rightarrow \cdot$, that $M = M' \in A$; otherwise, if $\Gamma \Rightarrow \Delta, x \in B$, it is to know

$$|_{N}\, \Delta \gg [N/x]M = [N/x]M' \in A\ (\Delta \gg N \in B)$$

and moreover, to know

$$|_{L,N}\, \Delta \gg [L/x]M = [N/x]M' \in A\ (\Delta \gg L = N \in B)$$

The simultaneous definition of multiple judgements may seem at first concerning, but it can be shown to be non-circular by induction on the length of the context Γ.

2. The definitions of types

We will now define the types of a simple computational type theory without universes. In the course of doing so, opportunities will arise for further clarifying the position of the judgements, meaning explanations and proofs on the one hand, and the propositions, definitions and verifications on the other hand.

2.1. The unit type. First, we introduce two canonical forms with trivial reduction rules:

(Canonical) $\qquad \mathsf{unit} \Rightarrow \mathsf{unit} \qquad \bullet \Rightarrow \bullet$

Next, we intend to make the judgement unit *type* evident; and this is done by defining what counts as a canonical verification of unit and when two such verifications are equal. To this end, we say that $\bullet$ is a canonical verification of unit, and that it is equal to itself. I wish to emphasize that this is the entire definition of the type: we have introduced syntax, and we have defined the canonical forms, and there is nothing more to be done.

In the presentations of type theory which are currently in vogue, a type is "defined" by writing out a bunch of inference rules, but in type theory, the definitions that we have given above are prior to the rules, which are justified in respect of the definitions and the meaning explanations of the judgements. For instance, based on the meaning of the various forms of sequent judgement, the following rule schemes

are justified:

$$\frac{}{\Gamma \gg \mathsf{unit}\ \mathit{type}} \qquad \frac{}{\Gamma \gg \mathsf{unit} = \mathsf{unit}\ \mathit{type}}$$

$$\frac{}{\Gamma \gg \bullet \in \mathsf{unit}} \qquad \frac{}{\Gamma \gg \bullet = \bullet \in \mathsf{unit}}$$

Each of the assertions above has evidence of a certain kind; since the justification of these rules with respect to the definitions of the logical constants and the meaning explanations of the judgements is largely self-evident, we omit it in nearly all cases. It is just important to remember that it is not the rules which define the types; a type A is defined in the course of causing the judgement $A\ \mathit{type}$ to become evident. These rules merely codify standard patterns of use, nothing more, and they must each be justified.

2.2. The empty type. The empty type is similarly easy to define. First, we introduce a constant:

(Canonical) $$\mathsf{void} \Rightarrow \mathsf{void}$$

To make the judgement $\mathsf{void}\ \mathit{type}$ evident, we will say that there are no canonical verifications of void, and be done with it. This definition validates some further rules schemes:

$$\frac{}{\Gamma \gg \mathsf{void}\ \mathit{type}} \qquad \frac{}{\Gamma \gg \mathsf{void} = \mathsf{void}\ \mathit{type}}$$

$$\frac{\Gamma \gg M \in \mathsf{void}}{\Gamma \gg \mathcal{J}}$$

The last rule simply says that if we have a verification of void, then we may conclude any judgement whatsoever. Remember that the inference rules are just notation for an *evident* hypothetical judgement, e.g. $\Gamma \gg \mathcal{J}$ ($\Gamma \gg M \in \mathsf{void}$).

Note that we did not introduce any special constant into the computation system to represent the elimination of a verification of void (in proof-theoretic type theories, this non-canonical form is usually called $\mathsf{abort}(R)$). This is because, computationally speaking, there is never any chance that we should ever have use for such a term, since we need only consider the evaluation of closed terms (which is guaranteed by the meaning explanations), and by its very definition, there can never be a closed verification of void.

2.3. The cartesian product of a family of types. This will be our first dependent type, and it will likewise be the first example of a non-trivial addition to the computation system. First, let us add our canonical and non-canonical forms

and their reduction rules:

$$\text{(Canonical)} \qquad \mathsf{\Pi}(A; x.B) \Rightarrow \mathsf{\Pi}(A; x.B) \qquad \lambda(x.E) \Rightarrow \lambda(x.E)$$

$$\text{(Non-canonical)} \qquad \frac{[M/x]E \Rightarrow N}{\mathsf{ap}(\lambda(x.E); M) \Rightarrow N}$$

This is the first time in this chapter that we have introduced a term former with binding structure; it should be noted that the intensional equality of expressions is up to alpha equivalence, and we will not pay attention to issues of variable renaming in our presentation.

We will make evident the following judgement scheme:

$$\frac{A\ \mathit{type} \quad x \in A \gg B\ \mathit{type}}{\mathsf{\Pi}(A; x.B)\ \mathit{type}}$$

Or, written as a hypothetical judgement:

$$\mathsf{\Pi}(A; x.B)\ \mathit{type}\ (A\ \mathit{type},\ x \in A \gg B\ \mathit{type})$$

This is to say, under the stated assumptions, we know what counts as a canonical verification of $\mathsf{\Pi}(A; x.B)$ and when two such verifications are equal. We will say that $\lambda(x.E)$ is a canonical verification of $\mathsf{\Pi}(A; x.B)$ just when we know $x \in A \gg E \in B$; moreover, that two verifications $\lambda(x.E)$ and $\lambda(y.E')$ are equal just when $z \in A \gg [z/x]E = [z/y]E' \in [z/x]B$.

By the meaning of the sequent judgement, this is to say that a lambda expression must be functional with respect to its domain (i.e. it must take equals to equals). We did not need to hypothesize directly two elements of the domain and their equality because this is part of the meaning explanation for the sequent judgement already. Likewise, two lambda expressions are equal when equal inputs yield equal results in both.

The familiar inference rules, which codify the standard mode of use for the family cartesian product, are justified by this definition:

$$\frac{\Gamma \gg A\ \mathit{type} \quad \Gamma, x \in A \gg B\ \mathit{type}}{\Gamma \gg \mathsf{\Pi}(A; x.B)\ \mathit{type}} \qquad \frac{\Gamma \gg A = A'\ \mathit{type} \quad \Gamma, z \in A \gg [z/x]B = [z/y]B'\ \mathit{type}}{\Gamma \gg \mathsf{\Pi}(A; x.B) = \mathsf{\Pi}(A'; y.B')\ \mathit{type}}$$

$$\frac{\Gamma, x \in A \gg E \in B}{\Gamma \gg \lambda(x.E) \in \mathsf{\Pi}(A; x.B)} \qquad \frac{\Gamma, z \in A \gg [z/x]E = [z/y]E' \in [z/x]B}{\Gamma \gg \lambda(x.E) = \lambda(y.E') \in \mathsf{\Pi}(A; x.B)}$$

$$\frac{\Gamma \gg M \in \mathsf{\Pi}(A; x.B) \quad \Gamma \gg N \in A}{\Gamma \gg \mathsf{ap}(M; N) \in [N/x]B} \qquad \frac{\Gamma \gg M = M' \in \mathsf{\Pi}(A; x.B) \quad \Gamma \gg N = N' \in A'}{\Gamma \gg \mathsf{ap}(M; N) = \mathsf{ap}(M'; N') \in [N/x]B}$$

Note that the type equality rule scheme that we gave above is structural; it is validated by the meaning explanations, but it is by no means the full totality of

possible equalities between family cartesian product types, which is extensional in this theory.

It will be instructive to explicitly justify the application rule above with respect to the meaning explanations, since I have claimed that such rules are posterior to the definitions we expounded prior to giving these rules.

PROOF. It suffices to consider the case that $\Gamma \Rightarrow \cdot$, because hypotheses may always be added to the context (this is called weakening). And so, by the meaning of the sequent judgement at the empty context, the rule amounts to the assertion

$$\mathsf{ap}(M; N) \in [N/x]B \ (M \in \Pi(A; x.B),\ N \in A)$$

By the meaning explanation for hypothetical judgement, and the definition of the family cartesian product type, we know that $M \Rightarrow \lambda(x.E)$ for some E such that we know $x \in A \gg E \in B$; from the meaning of the sequent judgement, we can conclude $|_L\, [L/x]E \in [L/x]B$ $(L \in A)$. On the other hand, from the computation rules, we know that if for some particular E', $[N/x]E \Rightarrow E'$, then we know $\mathsf{ap}(\lambda(x.E); N) \Rightarrow E'$; to demonstrate the evidence of the premise, we may instantiate L at N to know $[N/x]E \in [N/x]B$, whence by the meaning of membership, we know that there exists some canonical E' such that $[N/x] \Rightarrow E'$. □