A README ON CLASSICAL LOGIC - A FIRST ATTEMPT.

(CLASSICAL LOGIC WITHIN CONSTRUCTIVE CUBICAL AGDA)

ABSTRACT

Classical logic is compatible with constructive logic in the following sense: that it can be implemented within constructive logic. This idea is called double-negation translation and is due to Godel, Gentzen and Kuroda. Classical propositional and predicate logic is implemented here in Cubical Agda so that more insight can be gained and more theorems proven regarding the classical fragment. There are in effect two different concepts of both existence and universal quantification at play, the constructive and the classical, and these should both be available to the user of a proof assistant simultaneously and even in the same code; they are simply different, but both useful, and there is no need to treat classical and constructive logic as two different fields. It is merely a matter of precision that the user be clear about which definitions are being used at any one time in an application, proof or theorem statement.

INTRODUCTION

The implementation of a classical fragment within constructive logic is well-known: the so-called Godel fragment and the double-negation translations of Godel-Gentzen and Kuroda [5][6]. This is implemented here in Cubical Agda in such a way that more insight can be gained and more theorems proven regarding the classical fragment. Decidability is discussed. And classical ‘truth’ is defined simply as a non-falsifiability:

```
IsTrue : Type ℓ → Type ℓ
IsTrue A = ¬ ¬ A
```

Classical propositional calculus is then generated from the following logical operator:

```
Nor : (A : Type  ℓ) -> (B : Type  ℓ') -> Type (ℓ-max ℓ ℓ')
Nor A B = (¬ A) × (¬ B)
```

Some ‘classicality’ conditions are proven for each operator to show that what is being done makes classical sense. (Classical) Modus Ponens and classical propositional functionality are derived within the Type theory. Cubical equality is then shown to be able to define propositional equality. Some useful theorems are given in the code and a few are mentioned in the text: the excluded middle is easily proven, for example.

The predicate calculus is essentially reproduced by defining a single classical universal quantifier:

```
Forall : (A : Type ℓ) -> (B : A → Type ℓ') -> Type (ℓ-max (ℓ) (ℓ'))
Forall A B = (x : A) → IsTrue (B x)
```

This includes a classical existential quantifier (as per double-negation translation):

```
Exists : (A : Type ℓ) (B : A → Type ℓ') -> Type (ℓ-max (ℓ) (ℓ'))
```

Exists A B = IsTrue (Σ A B)

Using Cubical equality this can be shown to be constructible in terms of Forall. And any alternative formulations can likewise be shown to be equivalent. It ceases to matter exactly how such a fragment is constructed as it might in simpler Type Theories.

One inefficiency is encountered in the way operators can be defined to act on other operators. See the Discussion for a discussion of limitations.

Some axioms of predicate calculus due to Kleene are derived to show that we have in fact constructed a variety of classical logic.

CLASSICAL PROPOSITIONAL CALCULUS - THE GODEL FRAGMENT

In Cubical Agda the underlying logic of the Type theory is constructive or ‘intuitionistic’. One way to replicate classical propositional logic from within a constructive logic is to assume that ¬ ¬ P → P for any proposition. A similar result may also be achieved by assuming decidability of propositions as follows:

decEq→P : {P : Type ℓ} → Dec P → ¬ ¬ P → P

The condition that ¬ ¬ P → P is also equivalent to the excluded middle for propositions. This has to be applied a little carefully as it results that (¬ ¬ P) ≡ P only when P is a proposition in the Type theory sense of “isProp”, meaning in Cubical Agda terms:

```
isProp : Type ℓ → Type ℓ
isProp P = (x y : A) → x ≡ y
```

Thus, with this approach, we would need to assert both isProp P and decEq P for classical propositions; they must be both propositions in the Type theory sense and classical in the logic sense for the excluded middle to result. Fortunately we can always add decidability as a postulate consistently, since in Cubical Agda we have:

¬¬dec : {A : Type ℓ} → ¬ ¬ (Dec A)

But we do not have that all types are propositions in Cubical Agda. We may also not approve of assuming decidability when it may seem like too strong an additional assumption or postulate. Just as it may also seem to constructivists too strong an assumption to postulate the excluded middle, and potentially in some type theories these might even be false. In Cubical Agda, however, the following theorem in the code shows that ¬ ¬ P → P is non-falsifiable in Cubical Agda:

Non-Falsifiability-EM : {A : Type ℓ} -> (¬ ¬ ((¬ ¬ A) → A))

A more convincing way to proceed is to exploit the Godel fragment construction, which can be applied to any Type:

```
 -- IsTrue (defines classical 'truth')
IsTrue : Type ℓ → Type ℓ
IsTrue A = ¬ ¬ A
```

It simply entails creating a new proposition from any Type (or Sort in the Cubical Agda type system), by taking its double negation; by wrapping the would-be proposition in a double negation; that is, we take the non-falsifiability of the would-be proposition as the meaning of classical truth – by definition. This requires no assumptions, postulates or constraints to construct a classical proposition; it constructs them from Types. We get the following properties as a result:

isPropIsTrue : {A : Type ℓ} -> isProp (IsTrue A)

IsTrueClassical : {A : Type ℓ} -> IsTrue A ≡ (IsTrue (IsTrue A))

As a sanity check on consistency, we have:

IsTrue ⊥ : IsTrue ⊥ → ⊥
IsTrue ⊥ = λ z → z (λ z_1 → z_1)

isTrue ⊤ : Unit → IsTrue Unit
isTrue ⊤ tt = λ z → z tt

The classical propositional calculus can then be derived by adding the following generating function:

-- Nor
Nor : (A : Type ℓ) -> (B : Type ℓ') -> Type (ℓ-max ℓ ℓ')
Nor A B = (¬ A) × (¬ B)

And we can derive that this too is ‘classical’ with the following theorems (see code for proofs):

isPropNor : {A : Type ℓ} -> {B : Type ℓ'} → isProp (Nor A B)

NorClassical1 : {A : Type ℓ} -> {B : Type ℓ'} -> Nor A B ≡ (IsTrue (Nor A B))

NorClassical2 : {A : Type ℓ} -> {B : Type ℓ'} -> Nor A B ≡ (Nor (IsTrue A) B)

NorClassical3 : {A : Type ℓ} -> {B : Type ℓ'} -> Nor A B ≡ (Nor A (IsTrue B))

Unlike for IsTrue there are 3 required proofs of ‘classicality’ here. What is happening is that not only must NorClassical1 be satisfied, a sort of ‘external’ classicality condition, but also A and B can be any Types at all and still we get the same result as if A and B were classical. This can be termed the ‘internal’ classicality of Nor. The A and B don’t even need to be propositions. They are just Types in all their constructive Cubical complexity. For the purposes of classical propositional calculus we can consider them the required infinite list of symbols representing ‘propositions’.

To clarify what is happening here, consider the following theorems:

Nor≡nor : {A : Type ℓ} -> {B : Type ℓ'} → Nor A B ≡ (¬ (A ⊎ B))

Nor¬A¬B≡¬¬A×¬¬A : {A : Type ℓ} -> {B : Type ℓ'} → Nor (¬ A) (¬ B) ≡ (¬ ¬ A) × (¬ ¬ B)

We will define ‘classical’ for operators on propositions as any function on Types that is both internally and externally classical, or, by induction, constructed from (or cubically equal to, in the sense of A ≡ B) such classical operators.

For example,

IsFalse : (A : Type ℓ) → Type ℓ
IsFalse A = Nor A A

IsFalse≡¬ : {A : Type ℓ} -> (IsFalse A) ≡ (¬ A)

Thus the usual negation operator on types, of Cubical Agda, is itself proven to be ‘classical’. Further,

isPropIsFalse : {A : Type ℓ} -> isProp (IsFalse A)

Continuing in this constructive and inductive manner, all the other operators of classical propositional logic can be defined:

Or : (A : Type ℓ) -> (B : Type ℓ') -> Type (ℓ-max ℓ ℓ')
Or A B = ¬ (Nor A B)

Or≡¬Nor : {A : Type ℓ} -> {B : Type ℓ'} -> (Or A B) ≡ (¬ (Nor A B))

Or≡¬¬⊎ : {A : Type ℓ} -> {B : Type ℓ'} -> (Or A B) ≡ (¬ ¬ (A ⊎ B))

isPropOr : {A : Type ℓ} -> {B : Type ℓ'} -> isProp (Or A B)

And so on.

The ‘excluded middle’ follows:

EM : {A : Type ℓ} → (Or A (¬ A))

EM' : (A : Type ℓ) → IsTrue (A ⊎ (¬ A))

And continuing:

Nand : (A : Type ℓ) -> (B : Type ℓ') -> Type (ℓ-max ℓ ℓ')
Nand A B = ¬ (Nor (¬ A) (¬ B))

And : (A : Type ℓ) -> (B : Type ℓ') -> Type (ℓ-max ℓ ℓ')
And A B = ¬ (Nand A B)

And≡¬¬A×¬¬B : {A : Type ℓ} → {B : Type ℓ'} → And A B ≡ (¬ ¬ A) × (¬ ¬ B)

A classical implication can be defined (see code) and it is shown that:

A⇒B≡Or¬AB : {A : Type ℓ} -> {B : Type ℓ'} → (A ⇒ B) ≡ (Or (¬ A) B)

It follows from the preceding that a double implication is also classical:

_⟺_ : (A : Type ℓ) -> (B : Type ℓ') -> Type (ℓ-max ℓ ℓ')
A ⟺ B = ((A ⇒ B) × (B ⇒ A))

For example, we have a proof of:

A⟺B≡IsTrueA⇒B×IsTrueB⇒A : {A : Type ℓ} -> {B : Type ℓ'} -> (A ⟺ B) ≡ ((IsTrue (A ⇒ B)) × (IsTrue (B ⇒ A)))

And it is inevitable that the other ‘classicality’ conditions in the code result, including:

isProp⟺ : {A : Type ℓ} -> {B : Type ℓ'} -> isProp (A ⟺ B)

As well as Modus Ponens:

ModusPonens⟹-classical : {A : Type ℓ} -> {B : Type ℓ'} -> A ⇒ (A ⇒ B) ⇒ B

And a number of variations are also given. Useful functions thus derived include:

A≡B→A⇒C→B⇒C : {A B : Type ℓ} → {C : Type ℓ'} -> A ≡ B → A ⇒ C → B ⇒ C

Just to tie the circle with the initial consideration on decidability, there is also obviously the following:

IsTrueDecA : {A : Type ℓ} → IsTrue (Dec A)

That is, from within the Godel fragment, propositions are indeed decidable in some sense; decidability is not falsifiable constructively within Cubical Agda. Does that make it classically the case that all propositions are decidable? I propose that it *is* in the sense that what this is telling us is that there is a limit to what sort of propositions a Type or Sort in such a Type theory can represent. And they are as such all classically ‘decidable’ (in being either true or false classically – no excluded middle) even if a constructive proof of such does not exist in Cubical Type theory, that is, they need not be constructively decidable via any algorithm within the Type theory, but that they are either IsTrue or IsFalse *classically* remains the condition of classical logic – whether we have any such proofs or not.

Further, we may define a classical propositional equality and a classical propositional extensionality as follows:

```
_===_ : (p q : Type ℓ) → Type (ℓ-suc ℓ)
_===_ p q = IsTrue ((IsTrue p) ≡ (IsTrue q))
```

We can simplify this using the machinery of Cubical Agda in a powerful way:

```
_==_ : (p q : Type ℓ) → Type (ℓ-suc ℓ)
_==_ p q = (IsTrue p) ≡ (IsTrue q)
```

And the above two definitions are cubically equal:

p===q≡p==q : (p q : Type ℓ) → (p === q) ≡ (p == q)

We thus have classical propositional logic as an important fragment, a Godel fragment, within constructive Cubical Agda. Cubical logic enables the equivalence of operators, functions and types to be handled more easily than earlier type theories and more proofs can be more naturally presented. The concept of two classical propositions is clarified by the above – it *is* a cubical equality!

THE CLASSICAL PREDICATE CALCULUS

The preceding approach is now extended to the classical predicate calculus, as per the Godel-Gentzen and Kuroda double-negation translations.

As with classical propositional calculus, a version of classical predicate calculus can be implemented by adding very few extra functions; one in fact: a classical universal operator:

```
Forall : (A : Type ℓ) -> (B : A → Type ℓ') -> Type (ℓ-max (ℓ) (ℓ'))
Forall A B = (x : A) → IsTrue (B x)
```

We have ‘classicality’ as follows:

```
isPropForall : {A : Type ℓ} → {B : A → Type ℓ'} → isProp (Forall A B)

ForallClassical1 : {A : Type ℓ} → (B : A → Type ℓ') → Forall A B ≡ IsTrue (Forall A B)

ForallClassical2 : {A : Type ℓ} → (B : A → Type ℓ') → Forall A B ≡ (Forall A (λ x → IsTrue (B x)))
```

The second of these is a little unexpected, as the IsTrue does not contain the λ x part. The ‘x’ of course is not playing the role of a proposition; the resulting predicate B x is doing so however.

We notice here the first difficulty. Negation cannot be applied to B alone (due to a certain lack of polymorphism in the Type theory), as it is not a ‘Sort’ in Agda’s type system. We must apply negation to (B x) as a whole. We may define the following in the obvious way:

```
Forall¬ : (A : Type ℓ) -> (B : A → Type ℓ') -> Type (ℓ-max (ℓ) (ℓ'))
Forall¬ A B = (x : A) → ¬ (B x)
```

But we would have to define a more complex function to apply the negation directly to B. Such as:

```
UnaryOn→ : {A : Type ℓ} → (unary : (Type ℓ' → Type ℓ")) → (B : A → Type ℓ') → (A → Type ℓ")
UnaryOn→ {ℓ} {ℓ'} {ℓ"} {A} unary B = λ x → unary (B x)

-- And the analog of ¬ ...
IsFalseOn→ : {A : Type ℓ} → (B : A → Type ℓ') -> (A → Type ℓ')
IsFalseOn→  {ℓ} {ℓ'} {A} B = (UnaryOn→ IsFalse) B
```

This is somewhat annoying and makes for the need for extra code, further, it is conceptually confusing. Paper-based classical logic, on the other hand, allows this effortlessly.

In any case, we proceed:

```
isPropForall¬ : {A : Type ℓ} -> {B : A → Type ℓ'} -> isProp (Forall¬ A B)
```

And ‘classicality’ is proven as follows:

```
Forall¬→Forall'¬ :  {A : Type ℓ} -> {B : A → Type ℓ'} -> (Forall¬ A B) → Forall A (λ a -> ¬ (B a))
```

Similarly we can define a classical ‘Exists’:

Exists : (A : Type ℓ) (B : A → Type ℓ') -> Type (ℓ-max (ℓ) (ℓ'))
Exists A B = IsTrue (Σ A B)

And its ‘classicality’ may be proven by its construction in terms of ‘Forall’:

Exists≡¬Forall¬ : (A : Type ℓ) (B : A → Type ℓ') -> (Exists A B) ≡ (¬ (Forall¬ A B))

We have:

Exists¬ : (A : Type ℓ) (B : A → Type ℓ') -> Type (ℓ-max (ℓ) (ℓ'))
Exists¬ A B = ¬ ¬ (Σ A (λ a → ¬ (B a)))

And its ‘classicality’:

Exists¬≡ : {A : Type ℓ} {B : A → Type ℓ'} -> (Exists¬ A B) ≡ (Exists A (λ a -> ¬ (B a)))

Some basic theorems result:

Exists¬≡¬Forall : (A : Type ℓ) (B : A → Type ℓ') -> (Exists¬ A B) ≡ (¬ (Forall A B))

IsTrueA→ForallA→ : {A : Type ℓ} {B : A → Type ℓ'} -> IsTrue A → Forall A B → Exists A B

At this point we should ask: does our construction in terms of IsTrue, Nor and Forall really constitute a classical theory of logic, albeit likely a limited version thereof?

IS THIS REALLY A FORM OF CLASSICAL LOGIC?

With reference to the following books [1][2[[3][4]:

We show that what we have is indeed a classical predicate calculus. By starting with Kleene's axioms of propositional and predicate calculus we can establish this theorem.

Here are the proofs (they can be investigated in the code). In particular we now show the rules of propositional and predicate calculus as per [2] section 19, page 82. (Using automated proof search where possible):

```
-- Group A1 Postulates for the propositional calculus.
rule-1a : {A : Type ℓ} → {B : Type ℓ'} -> A ⇒ (B ⇒ A)
rule-1a = λ z z₁ → z₁ (λ z₂ z₃ → z₂ (λ _ → z z₃))

rule-1b : {A : Type ℓ} → {B : Type ℓ'} → {C : Type ℓ''} -> (A ⇒ B) ⇒ ((A ⇒ (B ⇒ C)) ⇒ (A ⇒ C))
rule-1b = λ z z₁ → z₁ (λ z₂ z₃ → z₃ (λ z₄ z₅ → z₄ (λ z₆ → z₂ (λ z₇ → z₇ (λ z₈ → z₈ z₆)
                (λ z₈ → z₈ (λ z₉ → z (λ z₁₀ → z₁₀ (λ z₁₁ → z₁₁ z₆) z₉)) z₅)))))

rule-2 : {A : Type ℓ} → {B : Type ℓ'} -> A ⇒ (A ⇒ B) ⇒ B
rule-2 = λ z z₁ → z₁ (λ z₂ z₃ → z₂ (λ z₄ → z₄ z z₃))
```

Rule 2 is written in the format of natural deduction in Kleene, so the following is included here. Note that the inference remains classical due to the ⇒

```
rule-2-uncurry : {A : Type ℓ} → {B : Type ℓ'} -> (A × (A ⇒ B)) ⇒ B
rule-2-uncurry {ℓ} {ℓ'} {A} {B} = classical-curry ModusPonens⟹-classical

rule-2-classicalB-hlp : {A : Type ℓ} → {B : Type ℓ'} -> (A × (A ⇒ B)) → IsTrue B
rule-2-classicalB-hlp aab = λ x → proj₂ aab (A¬¬A (proj₁ aab)) x
```

The following is that which best represents Kleene's rule-2. There is, however, an issue; we must assume B is 'classical' (to the extent that we need 'B ≡ IsTrue B'). Rule-2 is therefore conditional on an extra requirement beyond that of just B being a Type, it requires in effect an inductive base case in terms of the classicality of B:

```
rule-2-classicalB : {A : Type ℓ} → {B : Type ℓ'} -> (B ≡ IsTrue B) → (A × (A ⇒ B)) → B
rule-2-classicalB B-classical aab = transport (sym B-classical) (rule-2-classicalB-hlp aab)

rule-3 : {A : Type ℓ} → {B : Type ℓ'} -> (A ⇒ (B ⇒ (And A B)))
rule-3 = λ z z₁ → z₁ (λ z₂ z₃ → z₃ (λ z₄ → z₄ ((λ x → z₂ (λ _ → z x)) , z₂)))

rule-4a-hlp : {A : Type ℓ} → {B : Type ℓ'} -> ((¬ ¬ A) × (¬ ¬ B)) ⇒ A
rule-4a-hlp {A = A} {B = B} = λ x y → ((¬¬¬A¬A (¬¬AB→nnA x)) y)

rule-4b-hlp : {A : Type ℓ} → {B : Type ℓ'} -> ((¬ ¬ A) × (¬ ¬ B)) ⇒ B
rule-4b-hlp {A = A} {B = B} = λ x y → ((¬¬¬A¬A (¬¬AB→nnB x)) y)

rule-4a : {A : Type ℓ} → {B : Type ℓ'} -> (And A B) ⇒ A
rule-4a {ℓ}{ℓ'} {A} {B} = A≡B→A⇒C→B⇒C (sym And≡¬¬A×¬¬B) rule-4a-hlp
```

rule-4b : {A : Type ℓ} → {B : Type ℓ'} -> (And A B) ⇒ B
rule-4b {ℓ}{ℓ'} {A} {B} = A≡B→A⇒C→B⇒C (sym And≡¬¬A×¬¬B) rule-4b-hlp

rule-5a : {A : Type ℓ} → {B : Type ℓ'} → (A ⇒ (Or A B))
rule-5a {A = A} {B = B} = transport (sym A⇒B=IsTrueA→IsTrueB) λ a → A¬¬A (mkOr1 a)

rule-5b : {A : Type ℓ} → {B : Type ℓ'} → (B ⇒ (Or A B))
rule-5b = transport (sym A⇒B=IsTrueA→IsTrueB) λ a → A¬¬A (mkOr2 a)

rule-6 : {A : Type ℓ} → {B : Type ℓ'} → {C : Type ℓ"} -> (A ⇒ C) ⇒ ((B ⇒ C) ⇒ ((Or A B) ⇒ C))
rule-6 = λ z z_1 → z_1 (λ z_2 z_3 → z_3 (λ z_4 z_5 → z_4 (λ z_6 → z_6
((λ x → z_2 (λ z_7 → z_7 (λ _ → z (λ z_8 → z_8 (λ z_9 → z_9 x) z_5)) z_5)) ,
(λ x → z_2 (λ z_7 → z_7 (λ z_8 → z_8 x) z_5))))))

rule-7 : {A : Type ℓ} → {B : Type ℓ'} → (A ⇒ B) ⇒ ((A ⇒ (¬ B)) ⇒ (¬ A))
rule-7 = λ z z_1 → z_1 (λ z_2 z_3 → z_3 (λ z_4 → z_2 (λ z_5 →
z_5 (λ z_6 → z_6 z_4) (λ z_6 → z (λ z_7 → z_7 (λ z_8 → z_8 z_4) z_6)))))

rule-8 : {A : Type ℓ} → (¬ ¬ A) ⇒ A
rule-8 = λ z z_1 → z (λ z_2 → z_2 z_1)

-- Group A2 of Kleene [2] (postulates of predicate calculus)

Variables are 'free' and go in the 'context'. If there is an inference line, then variables must go in the correct context (ie above or below the inference line). Here it is on the left of the main constructive implication, meaning above the inference line. The inference line here is a constructive implication, the normal Type theory function arrow:

rule-9 : (X : Type ℓ) -> (A : X → Type ℓ') -> (C : Type ℓ") → (∀ (x : X) → (C ⇒ (A x))) → (C ⇒ (Forall X A))
rule-9 = λ X A C z z_1 z_2 → z_2 (λ x z_3 → z_1 (λ z_4 → z x (λ z_5 → z_5 z_4) z_3))

A variant where the variable x is not in the context as per the translation of Kleene's definition, but it still parses, since x is not 'contained' in C:

rule-9' : (X : Type ℓ) -> (A : X → Type ℓ') -> (C : Type ℓ") → (C ⇒ (∀ (x : X) → (A x))) → (C ⇒ (Forall X A))
rule-9' = λ X A C z z_1 z_2 → z_2 (λ x z_3 → z_1 (λ z_4 → z (λ z_5 → z_5 z_4) (λ z_5 → z_3 (z_5 x))))

In the following we have that t is a variable which is "free for x in A(x)" [Kleene] (ie not contained in A or x). This means Forall X A and A are not defined in terms of t. Despite t being in the context this is so, since X and A are given prior to t in the function definition.

rule-10 : (X : Type ℓ) -> (A : X → Type ℓ') -> ∀ (t : X) → ((Forall X A) ⇒ (A t))
rule-10 = λ X A t z z_1 → z (λ z_2 → z_2 t z_1)

As with rule-10, t is not 'contained' in A or x, though of course there is no 'x' here as such; it means that Exists X A is independent of t, which, despite t being in the context, it is (again X and A are given prior to t):

rule-11 : (X : Type ℓ) -> (A : X → Type ℓ') -> ∀ (t : X) → ((A t) ⇒ (Exists X A))
rule-11 = λ X A t z z_1 → z_1 (λ z_2 → z (λ z_3 → z_2 (t , z_3)))

```
rule-12-hlp2 : {A : Type ℓ} -> {B : Type ℓ'} -> {C : Type ℓ''} → IsTrue A → IsTrue B → (A -> B -> C) -> IsTrue C
rule-12-hlp2 = λ z z₁ z₂ z₃ → z₁ (λ z₄ → z (λ z₅ → z₃ (z₂ z₅ z₄)))

rule-12-hlp2' : {A : Type ℓ} -> {B : Type ℓ'} -> {C : Type ℓ''} → IsTrue A → IsTrue B → (A -> B ⇒ C) ⇒ IsTrue C
rule-12-hlp2' = λ z z₁ z₂ z₃ → z₃ (λ z₄ → z₂ (λ z₅ → z₁ (λ z₆ → z (λ z₇ → z₅ z₇ (λ z₈ → z₈ z₆) z₄))))

rule-12-hlp2'' : {A : Type ℓ} -> {B : Type ℓ'} -> {C : Type ℓ''} → IsTrue A → IsTrue B → (A -> B ⇒ C) → IsTrue C
rule-12-hlp2'' = λ z z₁ z₂ z₃ → z₁ (λ z₄ → z (λ z₅ → z₂ z₅ (λ z₆ → z₆ z₄) z₃))

rule-12-hlp2''' : {A : Type ℓ} -> {B : Type ℓ'} -> {C : Type ℓ''} → IsTrue A → IsTrue B → (A ⇒ B ⇒ C) → IsTrue C
rule-12-hlp2''' = λ z z₁ z₂ z₃ → z₂ (λ z₄ → z₁ (λ _ → z z₄)) (λ z₄ → z₄ z₁ z₃)

rule-12-hlp3 :  (X : Type ℓ) -> (A : X → Type ℓ') -> (C : Type ℓ'') → (Σ X A) → (((x : X) → A x ⇒ C) ⇒ C)
rule-12-hlp3 X A C sig a =   IsTrueA→B→IsTrueA→IsTrueB lemma4 (λ z → z (snd sig))
                        where
                          IsTruePushed :  (x : X) → IsTrue ((A x) ⇒  C)
                          IsTruePushed = IsTrueA→B→A→IsTrueB-dep2' a
                          lemma2 : IsTrue ((A (fst sig)) ⇒ C)
                          lemma2 = IsTruePushed (fst sig)
                          lemma3 : (IsTrue ((A (fst sig)) ⇒ C)) ≡ (IsTrue ((A (fst sig)) → C))
                          lemma3 = IsTrueA⇒B=IsTrueA→B-dep {X = X} {A = A} {B = C} (fst sig)
                          lemma4 : IsTrue ((A (fst sig)) -> C)
                          lemma4 = transport lemma3 lemma2
```

Here 'x' is in the context of the hypothesis to the inference:

```
rule-12 : (X : Type ℓ) -> (A : X → Type ℓ') -> (C : Type ℓ'') → (∀ (x : X) → ((A x) ⇒ C)) → ((Exists X A) ⇒ C)
rule-12 X A C Ax y = istC
                  where
                    existsXA : Exists X A
                    existsXA = transport (sym (ExistsClassical1 X A)) y
                    existsXAΣ : IsTrue (Σ X A)
                    existsXAΣ = existsXA
                    istxAC : IsTrue ((x : X) → A x ⇒ C)
                    istxAC = A¬¬A Ax
                    istC : IsTrue C
                    istC = rule-12-hlp2'' existsXAΣ istxAC (rule-12-hlp3 X A C)
```

These proofs require some explanation. In particular note that the natural deduction formulae (2, 9 and 12 in Kleene) here use constructive logic. That is, the combination of two inputs to an inference in natural deduction are taken to be constructive ‘and’, and the inference of natural deduction taken to be constructive ‘implication’. Further, ‘variables’ are taken to be constructive ‘foralls’ in the context of the formula, or in the context of the line in which it is used in a natural deduction definition. We have chosen here to use the Type theory as a meta-theory for the classical calculus.

But all the proofs result.

Taking B to be ‘classical’ in rule-2-classicalB is analogous to a base case of an induction, an induction that proves in what way this fragment is classical. Yet it is also constructive. There is no absolute barrier between the two.

DISCUSSION

A version of classical propositional and predicate calculus, that is classical logic, has been implemented in Cubical Agda.

There is clearly a limit to what can be expressed via Types/Sorts following the approach here. Each Type theory has its own limits as to what can be constructed. For example, in terms of Types, often we are limited to inductive or co-inductive Types. Not every Type theory facilitates impredicative propositions, for example, Cubical Agda included. Nevertheless, we have given an account of the classical propositional and predicate calculus as definable within Cubical Agda. It is a limited implementation.

We may ask: “What exactly are the limits on types in expressing classical logical statements in terms of the classical “IsTrue” fragment?” Much of this may depend on the specifics of the Type theory as just mentioned. Such issues as: how to construct a set theory, what kinds of data types are definable, how are we to define real numbers, the axiom of choice, and so on, are not attempted. This is just a first attempt at implementing classical logic in Cubical Agda.

Type theories can always seemingly be added to and enriched with features, such as in Cubical Agda we have the Higher Inductive Types, or in Coq we have impredicative Prop. The limits of what can be done with a constructive version of classical logic is limited by the Type theory it is constructed in. Perhaps there is ultimately no limit to how much of classical logic and mathematics can be encoded in constructive type theories? Exactly what constructions are presently possible in Cubical Agda, or related Type theories, is not clear.

An equality on classical propositions was given in a natural way in terms of Cubical equality. It *is* just their cubical equality!

A further issue that is sometimes confusing, as introduced at the beginning, is the role of decidability, and its non-falsifiability: you cannot construct a proof that a proposition is not decidable in Cubical Agda, if using just Cubical Agda. And this leads to the classical decidability of all propositions in the classical fragment of Cubical Agda. However, this classical decidability doesn’t mean that such a proposition is decidable in the constructive sense, it rather means the excluded middle applies and it must be either IsTrue or IsFalse whether we can construct a classical proof to that effect or not. Whilst this observation will not effect most of classical logic, or even much of classical mathematics necessarily, it is necessary to deal with this sort of issue.

In the preceding one particular annoyance (or limitation) arose. The problem of making classical operators *polymorphic* across more than just Types and Sorts. The issue was constructing a ‘Not’ operator on B. B, unfortunately is not a Type or Sort, but a Term:

```
IsFalseOn→ : {A : Type ℓ} → (B : A → Type ℓ') -> (A → Type ℓ')
IsFalseOn→  {ℓ} {ℓ'} {A} B = (UnaryOn→ IsFalse) B
```

The trouble is that the theorems proven on IsFalse do not apply automatically to IsFalseOn→. The relationships to other operators has to be re-proven or some way to carry the proofs over needs to be constructed. This is not necessarily critical, but it is a nuisance, and limits what can be done naturally with such a Type theory and makes this implementation inadequate. The idea would be to be able to define the operator ‘Not’ more generically. But not only ‘Not’, there is a need for more power here.

Using a sort of pseudo-code variant of Type theory to express this, what would be nice would be a way to define something like this:

```
¬' : (B : Type  ℓ) -> Type ?
¬' B = B ->  ⊥

∀¬ : {A : Type  ℓ} -> (B : A → Type ℓ') →  (A → Type ?)
∀¬ {ℓ} {ℓ'} {A} B a = ¬' (B a)

∀∀¬ : {A : Type  ℓ} -> (B : A -> A → Type ℓ') →  (A → A → Type ?)
∀∀¬ {ℓ} {ℓ'} {A} B a =  ∀¬ (B a)

∀∀∀¬ : {A : Type  ℓ} -> (B : A -> A -> A → Type ℓ') →  (A → A -> A → Type ?)
∀∀∀¬ {ℓ} {ℓ'} {A} B a =  ∀∀¬ (B a)
```

...using some more concise construction, like this:

```
¬' : (B : Type ℓ) -> Type ℓ
¬' (B : ?) = B ->  ⊥
¬' (B : A -> Type ℓ) = λ a ->  ¬' (B a)
```

If this was possible then such a nuisance might be avoided and the classical fragment of Type theories could be brought closer in line with ‘paper’ classical logic. Whilst this may or may not be feasible without causing other problems in the Type theory, nevertheless some better way to solve the above problem within Type theories would be useful to address. Agda Zulip chat recommended the use of Nary functions. This was something of a can of worms and remains a work in progress.

Two easily automated proofs of the classical drinker paradox [9] are given and analogous formulation (simply by way of comparison) given using only constructive equivalents. As a theorem of classical logic it cannot be proven for constructive logic. But the constructive formulation is given here for comparison to show that classical and constructive logic can be used side-by-side. It is given as a type, and so the statement of the theorem, at least, can be expressed in Cubical Agda.

Agda’s automated proof search was shown to be just as useful in doing classical logic as the constructive logic to which it is more usually applied.

REFERENCES
---

[1] D. Hilbert, P. Bernays, "Grundlagen der Mathematik", 1–2 , Springer (1934–1939)
[2] S.C. Kleene, "Introduction to metamathematics", North-Holland (1951)
[3] P.S. Novikov, "Elements of mathematical logic", Oliver & Boyd (1964) (Translated from Russian)
[4] G. Takeuti, "Proof theory", North-Holland (1975)
[5] K. Gödel (1933), "Zur intuitionistischen Arithmetik und Zahlentheorie", Ergebnisse eines mathematischen Kolloquiums, v. 4, pp. 34–38 (German) (1933). Reprinted in English translation as "On intuitionistic arithmetic and number theory" in The Undecidable, M. Davis, ed., pp. 75–81.
[6] G. Gentzen, "Die Widerspruchfreiheit der reinen Zahlentheorie", Mathematische Annalen, v. 112, pp. 493–565 (German) (1936). Reprinted in English translation as "The consistency of arithmetic" in The collected papers of Gerhard Gentzen, M. E. Szabo, ed.
[7] “Homotopy type theory: Univalent foundations of mathematics.” The Univalent Foundations Program, Institute for Advanced Study (2013)
[8] The Agda Proof Assistant, https://agda.readthedocs.io/en/v2.6.3/index.html
[9] The Cubical Agda Proof Assistant, https://agda.readthedocs.io/en/v2.6.3/language/cubical.html
[10] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. Proc. ACM Program. Lang. 3, ICFP, Article 87 (August 2019), 29 pages. https://doi.org/10.1145/3341691
[11] R.Smullyan. “What is the Name of this Book? The Riddle of Dracula and Other Logical Puzzles.” Prentice Hall. Chapter 14. How to Prove Anything. *The Drinking Principle. pp. 209-211*. (1978)