Chapter 8

Deduction

So far I have introduced the rules of Type Theory informally and have explained logical principles by the proposition as types explanation. But in the end we want to make the rules of reasoning precise. This can be done by presenting a formal system defining the logic or the type theory.

In this chapter we will look at a very modest subset, basically the logic corresponding to propositional logic using only implication, which corresponds to section 2.1. We are not going to model β-reduction but only present a notion of provability which is called *natural deduction*. We will also study the relationship to combinatory logic and give a formal account of the translation presented in section 2.4.

Another import aspect of a logical system is its semantics. We will use the semantics to prove non-derivability results, i.e. that something is not provable. First we use the naive semantics which allows us to disprove statements by giving a counterexample but then we will explore a more precise semantics, Kripke models, which allows us to disprove also classically valid statements like Pierce's formula $((\mathsf{P} \to \mathsf{Q}) \to \mathsf{P}) \to \mathsf{P}$. We prove completeness of Kripke models (i.e. everything which holds semantically is also derivable) and then use a modified from of the completeness result to prove normalisation — this is inspired by a technique called*normalisation by evaluation*.

This chapter also serves a different purpose in giving more examples how the technology we have introduced in the previous chapters can be used, in particular how we can use inductive families to present deductive systems.

8.1 The rules of natural deduction

The first step is to formally define the syntax of propositions, i.e. formulas: no surprises here they will be trees. The trees are built from implications which we will write as _[⇒]_ but what is at the leaves of the trees? OK we need to introduce propositional variables which I am going to represent as strings.

```
data Form : Set where
  Var : String → Form
  _[⇒]_ : Form → Form → Form
```

So for example we can represent the proposition

$$(\mathsf{Q} \to \mathsf{R}) \to (\mathsf{P} \to \mathsf{Q}) \to \mathsf{P} \to \mathsf{R} : \mathsf{prop}$$

as

```
((Var "Q") [⇒] (Var "R")) [⇒] ((Var "P") [⇒] (Var "Q"))
  [⇒] (Var "P") [⇒] (Var "R")
```

Using strings to represent propositional variables is not the best choice but we will go for it in the moment. However, we will not use strings for the variables used in terms or derivations as we will see in a moment.

We want to define a predicate what propositional formulas are derivable, that is ⊢ : Form → Set but this is not sufficient because we need to take care of assumptions as well. That is we are going to introduce a type Con : Set of contexts and then define derivations as _⊢_ : Con → Form → Set.

How to represent contexts? As I said I am not going to use names to represent variables but only pointers, or actually a form of *de Bruijn indices.* Hence a context is just a sequence of assumptions, i.e. a list of formulas:

```
Con = List Form
```

We are going to use $\Gamma\ \Delta\ \Theta$: Con to quantify over contexts and A B C : Form for formulas.

Ok, what are the rules of natural deduction. There are two rules to deal with implication: the first one corresponds to λ-abstraction and it conveys the idea: to prove $A \Rightarrow B$ we assume A and prove B:

$$\frac{(\mathsf{A} :: \Gamma) \vdash \mathsf{B}}{\Gamma \vdash \mathsf{A}\,[\Rightarrow]\,\mathsf{B}}\ \mathsf{lam}$$

The other rule corresponds to application: If we know A ⇒ B and also A then we can derive B. The context stays the same, i.e. the number of assumptions don't change.

$$\frac{\Gamma \vdash \mathsf{A}\,[\Rightarrow]\,\mathsf{B} \quad \Gamma \vdash \mathsf{A}}{\Gamma \vdash \mathsf{B}}\ \mathsf{app}$$

This rule is also called *modus ponens.*

We still need some structural rules to access the context of assumptions. In the usual syntax of λ-calculus we use named variables but really these variables are just pointers into the context. Hence we can replace variables by numbers representing the offset within the context. This was introduced by Niklas de Bruijn and hence we call this representation of variables *de Bruijn indices.* E.g. the λ-term λ f → λ x → f x x would be written as $\lambda\ \lambda$ 1 0 0, because x is represented by $\mathsf{0}$ since it was the last variable introduced and f is represented by the index 1 because it is one step down in our list of assumptions.

Hence we introduce the two rules:

$$\frac{}{(\mathsf{A} :: \Gamma) \vdash \mathsf{A}}\ \mathsf{zero} \qquad \frac{\Gamma \vdash \mathsf{A}}{(\mathsf{B} :: \Gamma) \vdash \mathsf{A}}\ \mathsf{suc}$$

zero allows us to access the last item in the context, and suc discards the last item and allows us to access previous items. So for example in the context

```
(Var "P") :: (Var "Q") :: []
```

We can access

$$\frac{}{(\mathsf{Var}\ \texttt{"P"}) :: (\mathsf{Var}\ \texttt{"Q"}) :: [] \vdash \mathsf{Var}\ \texttt{"P"}}\ \mathsf{zero}$$

and

$$\frac{\dfrac{}{(\mathsf{Var}\ \texttt{"Q"}) :: [] \vdash \mathsf{Var}\ \texttt{"Q"}}\ \mathsf{zero}}{(\mathsf{Var}\ \texttt{"P"}) :: (\mathsf{Var}\ \texttt{"Q"}) :: [] \vdash \mathsf{Var}\ \texttt{"Q"}}\ \mathsf{suc}$$

Actually since we don't have a special type of *assumptions* (corresponding to variables) we can apply suc to any derivation. This rule is usually called weakening. In terms of de Bruijn indices it means that we can replace 1 2 by suc (0 1) that is we *weaken* all free variables, i.e. add a successor.

As before for _⩽_, derivations are an indexed inductive definition with constructors corresponding to rules:

```
data _⊢_ : Con → Form → Set where
  zero : (A :: Γ) ⊢ A
  suc  : Γ ⊢ A → (B :: Γ) ⊢ A
  lam  : (A :: Γ) ⊢ B → Γ ⊢ (A [⇒] B)
  app  : Γ ⊢ (A [⇒] B) → Γ ⊢ A → Γ ⊢ B
```

As an example we derive (Q → R) → (P → Q) → P → R. That is we translate the meta level Agda construction

```
λ f g x → f (g x) : (Q → R) → (P → Q) → P → R
```

into

```
d-C : [] ⊢ ((Var "Q") [⇒] (Var "R")) [⇒] ((Var "P") [⇒] (Var "Q"))
         [⇒] (Var "P") [⇒] (Var "R")
d-C = lam (lam (lam (app (suc (suc zero)) (app (suc zero) zero))))
```

8.2 Semantics

Ok, we can translate λ-terms into derivations in our calculus, using de Bruijn indices to represent variables. However, certain formulas should not be derivable, for example we should not be able to derive

```
[] ⊢ ((Var "P") ⇒ (Var "Q")) ⇒ (Var "P").
```

Why not. Because we can actually show that this proposition is false for instances of the variables:

```
cex : ({P Q : prop} → (P ⇒ Q) ⇒ P) → ⊥
cex h = h {⊥} {⊤} (λ x → tt)
```

Hence if our system is reasonable, the technical term is *sound*, we should expect that this formula is not derivable.

When we needed to show that certain instances of _⩽_ are not derivable, e.g. ¬ (3 ⩽ 2) we could just use pattern matching which ended with impossible branches. Can we do the same here to prove:

```
([] ⊢ (((Var "P") [⇒] (Var "Q")) [⇒] (Var "P"))) → ⊥
```

We quickly notice that this doesn't lead anywhere. In particular the app-rule allows us to introduce more and more complicated formulas whose unprovability doesn't lead to a situation where no rules are applicable.

However, the solution is to actually prove soundness, that is to show that any formula we can derive is actually true for any instances of the propositional variables. This provides more than only just a tool to prove non-derivability it also shows us that our system is sensible.

We define semantics of formulas, contexts and derivations. The semantics of a formula is a proposition, but we need to know an interpretation of the propositional variables. I call this an *environment*:

```
Env = String → prop
```

Now we can define the semantics of formulas:

```
⟦_⟧F : Form → Env → prop
⟦ Var x ⟧F ρ       = ρ x
⟦ A [⇒] B ⟧F ρ = ⟦ A ⟧F ρ ⇒ ⟦ B ⟧F ρ
```

I am using here Scott brackets ⟦ .. ⟧, named after Dana Scott, to signal the demarcation between syntax and semantics.

The definition is very straightforward, variables we look up in the context and _[⇒]_ is translated as _⇒_. Contexts are just the conjunction of all the assumptions, hence:

```
⟦_⟧C : Con → Env → prop
⟦ [] ⟧C ρ       = ⊤
⟦ A :: Γ ⟧C ρ = ⟦ A ⟧F ρ ∧ ⟦ Γ ⟧C ρ
```

Finally, a derivation is translated into *the assumptions imply the conclusions:*

```
⟦_⟧d : Γ ⊢ A → {ρ : Env} → ⟦ Γ ⟧C ρ → ⟦ A ⟧F ρ
⟦ zero ⟧d (p , _)   = p
⟦ suc d ⟧d (_ , γ)  =  ⟦ d ⟧d γ
⟦ lam d ⟧d γ        =  λ p → ⟦ d ⟧d (p , γ)
⟦ app d e ⟧d γ      =  ⟦ d ⟧d γ (⟦ e ⟧d γ)
```

The rules for zero and suc translate indices into the access to the context (similar to our definition of _!!_ earlier), and lam is translated using λ-abstraction and app using application.

We can now reap the fruits of our efforts and apart from having the happy feeling that our calculus actually makes sense we can prove the non-derivability example. First we need to cook up an environment:

```
ρ0 : Env
ρ0 "P"  =  ⊥
ρ0 "Q"  =  ⊤
ρ0 _    =  ⊥
```

Here we also see why I don't like to use strings for variables: we need to assign a meaning to every string and not just to the ones we used in our formula. Here our solution is to interpret all other variables with ⊥ as a default.

We are now ready to show the non-derivability result:

```
cex-d : ([] ⊢ (((Var "P") [⇒] (Var "Q")) [⇒] (Var "P"))) → ⊥
cex-d d  =  ⟦ d ⟧d {ρ0} tt (λ ())
```

The method we have used exploits that we could refute the proposition semantically whose non-derivability we wanted to show. However, for a classically true formula we cannot do this, as we cannot refute excluded middle as we have seen previously. Hence the naive semantics we have shown here isn't good enough. While it is *sound*, it is not *complete*.

8.3 Equivalence with combinatory logic

In section 2.4 we informally showed that pure functions in simply typed λ-calculus can be represented using just the combinators S and K. We will do this formally now and show that provability in combinatory logic, i.e. a logic which uses only S, K and application but no lam is equivalent to provability in natural deduction.

One of the main advantages of combinatory logic is that we don't need to use variables, aka assumptions. That means we don't have to use contexts but can use a one sided version of ⊢:

```
data ⊢sk : Form → Set where
  SS : ⊢sk ((A [⇒] B [⇒] C)
              [⇒] (A [⇒] B) [⇒] A [⇒] C)
  KK : ⊢sk (A [⇒] B [⇒] A)
  app : ⊢sk (A [⇒] B) → ⊢sk A → ⊢sk B
```

Our goal is to show that propositions provable from no assumptions are equivalent to propositions provable in propositional logic.

```
thm : ([] ⊢ A) ⇔ ⊢sk A
```

The direction from right to left is the easy one. All we need to show is that the combinators S and K are provable in natural deduction:

```
K-d : [] ⊢ (A [⇒] B [⇒] A)
K-d = lam (lam (suc zero))
S-d : [] ⊢ ((A [⇒] B [⇒] C)
  [⇒] (A [⇒] B) [⇒] A [⇒] C)
S-d =
  lam (lam (lam (app (app (suc (suc zero)) zero)
    (app (suc zero) zero))))
```

We only need them in the empty context here but we could actually prove them in any context. We can now show one implication ⊢sk A ⇒ [] ⊢ A by induction over the derivation of ⊢sk:

```
proj₂ thm SS = S-d
proj₂ thm KK = K-d
proj₂ thm (app d e) = app (proj₂ thm d) (proj₂ thm e)
```

The other direction is a bit more difficult. We cannot hope to prove anything by induction over [] ⊢ A because as soon as we encounter lam we need to talk about non-empty contexts.

To be able to compare natural deduction with combinatory logic we introduce a two-sided version of natural deduction with assumptions. Here we are using the same rules as for natural deduction:

```
data _⊢skC_ : Con → Form → Set where
  zero : (A :: Γ) ⊢skC A
  suc  : Γ ⊢skC A → (B :: Γ) ⊢skC A
  SS   : Γ ⊢skC ((A [⇒] B [⇒] C)
                [⇒] (A [⇒] B) [⇒] A [⇒] C)
  KK   : Γ ⊢skC (A [⇒] B [⇒] A)
  app  : Γ ⊢skC (A [⇒] B) → Γ ⊢skC A → Γ ⊢skC B
```

We can show that derivability in the empty context for ⊢skC coincides with provability in ⊢sk. Actually for our proof we only need one direction:

```
skC→sk : [] ⊢skC A → ⊢sk A
skC→sk SS        = SS
skC→sk KK        = KK
skC→sk (app d e) = app (skC→sk d) (skC→sk e)
```

Unlike for natural deduction we can prove this by derivation over [] ⊢skC A for the empty context because there is no lam and the variable rules do not apply in the empty context.

We can now try to show a more general implication relating natural deduction (_⊢_) and the two-sided combinatory logic: (_⊢skC_):

```
⊢→⊢skC : Γ ⊢ A → Γ ⊢skC A
```

It is clear that we are going to prove this by induction over Γ ⊢ A, i.e. we pattern match on the derivation:

```
⊢→⊢skC zero        = ?
⊢→⊢skC (suc d)     = ?
⊢→⊢skC (lam d)     = ?
⊢→⊢skC (app d e)   = ?
```

The cases for zero, suc and app are straightforward because we have corresponding constructors for ⊢skC:

```
⊢→⊢skC zero        = zero
⊢→⊢skC (suc d)     = suc (⊢→⊢skC d)
⊢→⊢skC (app d e)   = app (⊢→⊢skC d) (⊢→⊢skC e)
```

It remains the case for lam which is the core of the proof, namely that we can simulate lam using just S and K. Hence we need to prove a lemma, that _⊢skC_ is closed under lam:

```
lam-sk : (A :: Γ) ⊢skC B → Γ ⊢skC (A [⇒] B)
```

We will do this by induction over derivations again:

```
lam-sk zero        = ?
lam-sk (suc d)     = ?
lam-sk SS          = ?
lam-sk KK          = ?
lam-sk (app d e)   = ?
```

The first line, the case for zero we used zero to prove (A :: Γ) ⊢skC A hence we need to show Γ ⊢skC (A [⇒] A). This is exactly the point where we needed I. Hence we better derive it:

```
II : Γ ⊢skC (A [⇒] A)
II {A = A} = app (app SS KK) (KK {B = A})
```

Here we encounter the same problem as before, we need to instantiate one of the propositional variables explicitly. Using II we can cover the zero-case:

```
lam-sk zero = II
```

The next case is suc, given a derivation d : Γ ⊢skC A we get suc d : (B :: Γ) ⊢skC A. Hence we need to derive Γ ⊢skC B ⇒ A. In this case we used suc aka weakening to ignore the last variable and this is clearly the time for K:

```
lam-sk (suc d)  =  app KK d
```

The cases where we used SS or KK to derive (A :: Γ) ⊢skC B are similar, because we certainly ignored the last variable. Hence we are also using KK for those cases together with the same combinator:

```
lam-sk SS  =  app KK SS
lam-sk KK  =  app KK KK
```

The final and interesting case is app, we are abstracting from an application. That is exactly the point which justifies S which distributes abstractions over an application. This is the only case where we actually need to refer to the recursive result in the proof:

```
lam-sk (app d e)  =  app (app SS (lam-sk d)) (lam-sk e)
```

We have now covered all the cases:

```
lam-sk  :  (A :: Γ) ⊢skC B  →  Γ ⊢skC (A [⇒] B)
lam-sk zero         =  II
lam-sk (suc d)      =  app KK d
lam-sk SS           =  app KK SS
lam-sk KK           =  app KK KK
lam-sk (app d e)    =  app (app SS (lam-sk d)) (lam-sk e)
```

Now we can complete the proof of ⊢→⊢skC by filling in the case for lam:

```
⊢→⊢skC  :  Γ ⊢ A  →  Γ ⊢skC A
⊢→⊢skC zero         =  zero
⊢→⊢skC (suc d)      =  suc (⊢→⊢skC d)
⊢→⊢skC (lam d)      =  lam-sk (⊢→⊢skC d)
⊢→⊢skC (app d e)    =  app (⊢→⊢skC d) (⊢→⊢skC e)
```

We can now prove the other direction of thm, namely that [] ⊢ A ⇒ ⊢sk A by using ⊢→⊢skC in the special case of the empty context and combining it with skC→sk:

```
thm  :  ([] ⊢ A) ⇔ ⊢sk A
proj₁ thm d            =  skC→sk (⊢→⊢skC d)
proj₂ thm SS           =  S-d
proj₂ thm KK           =  K-d
proj₂ thm (app d e)    =  app (proj₂ thm d) (proj₂ thm e)
```

We can apply the left to right direction to our example to derive a combinatory proof of the example derivation:

```
d-C-sk : ⊢sk (((Var "Q") [⇒] (Var "R")) [⇒] ((Var "P") [⇒] (Var "Q"))
           [⇒] (Var "P") [⇒] (Var "R"))
d-C-sk = proj₁ thm d-C
```

The derivation we obtain is truly huge:

```
app
(app SS
(app (app SS (app KK SS))
  (app
    (app SS
      (app (app SS (app KK SS)) (app (app SS (app KK KK)) (app KK SS))))
    (app
      (app SS
        (app (app SS (app KK SS)) (app (app SS (app KK KK)) (app KK KK))))
      (app (app SS (app KK KK)) (app (app SS KK) KK))))))
(app
(app SS
  (app (app SS (app KK SS))
    (app
      (app SS
        (app (app SS (app KK SS)) (app (app SS (app KK KK)) (app KK SS))))
      (app
        (app SS
          (app (app SS (app KK SS)) (app (app SS (app KK KK)) (app KK KK))))
        (app (app SS (app (app SS (app KK SS)) (app KK KK)))
          (app KK KK))))))
(app
  (app SS
    (app (app SS (app KK SS))
      (app
        (app SS
          (app (app SS (app KK SS)) (app (app SS (app KK KK)) (app KK SS))))
        (app (app SS (app KK KK)) (app KK KK)))))
  (app (app SS (app KK KK)) (app KK KK))))
```

Indeed, we have implemented only the inefficient algorithm which is unsuitable when translating by hand. In section 2.4 we informally described a more efficient version which is more useful when doing the translation by hand. I leave it as an exercise to implement a modified version of lam-sk which implements the more efficient translation.

8.4 Kripke models

The naive semantics we introduced in section 8.2 isn't good enough to prove the non-derivability of propositions which are classically true such as the Pierce formula $((P \to Q) \to P) \to P$, this can be easily seen by checking the truth table.

It is less obvious that we cannot derive it. We can show that it entails the excluded middle

```
Pierce = {P Q : prop} → ((P → Q) → P) → P
```

```
pierce→tnd : Pierce → {P : prop} → TND P
pierce→tnd pierce {P} = pierce {TND P} {⊥} (λ p → case⊥ (nntnd p))
```

which strongly suggests that we cannot prove it but this doesn't amount to a proof.

To address this problem we need to come up with a better semantics, one which is actually complete that is for any underivable proposition we can construct a semantic counterexample. Or more positively: if a formula is true in the semantics then it is also derivable. This is the inverse of soundness which just stated that derivable propositions are true in the semantics.

A semantics which is complete for the fragment of intuitionistic propositional logic we have presented are Kripke models. The basic idea is that we have different worlds which correspond to stages of knowledge. In each world we know that certain atomic propositions hold and if we expand our knowledge we may learn that additional propositions are true but one which was true will remain so. The interesting point is how to interpret implication. We say that $\mathsf{P} \to \mathsf{Q}$ holds in a world if whenever in the future I will find out P then at this point Q will also hold.

Using this idea we can present a countermodel using just two worlds: lets call them w_0 and w_1. w_1 is a future of w_0, we write $\mathsf{w}_1 \sqsubseteq \mathsf{w}_0$ and while we know nothing in w_0, we know P in w_1, we write $\mathsf{w}_1 \Vdash \mathsf{P}$. Now we observe that $(\mathsf{P} \to \mathsf{Q}) \to \mathsf{P}$ holds in w_0, we write $\mathsf{w}_0 \Vdash (\mathsf{P} \to \mathsf{Q}) \to \mathsf{P}$ because in w_0 Q doesn't hold hence the implication holds trivially and in the future w_1 we actually know P hence the formula holds. However, since w_0 doesn't force P, the Pierce formula doesn't hold in w_0. Hence we have found a countermodel.

Ok, lets do this a bit more formally: A Kripke model is given by

```
record Kripke : Set₁ where
  field
    Worlds : Set
    _⊑_ : Worlds → Worlds → prop
    refl⊑ : {w : Worlds} → w ⊑ w
    trans⊑ : {w₀ w₁ w₂ : Worlds} → w₀ ⊑ w₁ → w₁ ⊑ w₂ → w₀ ⊑ w₂
```

Here _⊑_ is the future relation, we write $\mathsf{w}_1 \sqsubseteq \mathsf{w}_0$ to mean that w_1 is a future of of w_0. We assume that the future relation is reflexive and transitive, i.e. it is a preorder.

```
_⊩_ : Worlds → String → prop
mon⊩ : {w₀ w₁ : Worlds}
  → w₁ ⊑ w₀ → {x : String} → w₀ ⊩ x → w₁ ⊩ x
```

The relation $\mathsf{w} \Vdash \mathsf{x}$ (often called *forcing*) expresses that the propositional variable x presented by a string holds in world w. This should be monotone, any variable we know holds now will still be valid in the future.

This completes the definition of a Kripke model. Now we assume as given a Kripke model K : Kripke and we are going to extend _⊩_ to formulas and context and uses this to express what it means that a sequent Γ ⊢ A is valid in K. We define _⊩F_ using the interpretation of _[⇒]_ suggested earlier:

```
_⊩F_ : Worlds → Form → prop
w ⊩F Var x = w ⊩ x
w ⊩F A [⇒] B = {w' : Worlds} → w' ⊑ w → w' ⊩F A → w' ⊩F B
```

We extend _⊩_ to contexts, which simply says that it should hold for all formulas in the context. As in the naive semantics we are interpreting a context as the conjunction of all the formulas it contains.

```
_⊩C_ : Worlds → Con → prop
w ⊩C [] = True
w ⊩C (A :: Γ) = w ⊩F A ∧ w ⊩C Γ
```

We write Γ ⊨ A to express that semantically Γ entails A, i.e. every world that forces Γ also forces A:

```
_ ⊨ _ : Con → Form → prop
Γ ⊨ A = {w : Worlds} → w ⊩C Γ → w ⊩F A
```

We can now show soundness by induction over Γ ⊢ A:

```
⟦_⟧ : Γ ⊢ A → Γ ⊨ A
```

I leave the details as an exercise. To show soundness you first need to show that _⊩F_ and _⊩C_ are monotone.

Now to complete the proof of the non-derivability of the Pierce formula we have to formally define the Kripke model we sketched earlier.

```
module Pierce where
  data W : Set where
    w0 w1 : W
  data _⊑P_ : W → W → prop where
    refl : {w : W} → w ⊑P w
    w10 : w1 ⊑P w0
  data _⊩P_ : W → String → prop where
    w1p : w1 ⊩P "P"
```

I leave it as an exercise to complete the definition of the Kripke model. We can then show that the Pierce formula is not forced in w_0:

```
not⊩Pierce : ¬ (w0 ⊩F (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
```

and hence using soundness it is not provable:

```
no-Pierce-deriv : ¬ ([] ⊢ (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
```

8.5 Completeness

Ok, Kripke models are good enough to refute the Pierce formula but are they good enough to refute any formula that is not derivable? This means are they complete? Indeed we can construct one Kripke model, the universal model, which models exactly derivability. Hence if a formula is holds in all Kripke models then it will also hold in this one and hence it is derivable.

We construct this model from out derivation relation, that is we turn derivability into a Kripke model. The future relation we obtain be extending derivability on contexts, we define Γ ⊢C Δ, if all the formulas in Δ are derivable from Γ. Formally this can be defined as follows:

```
data _⊢C_ : Con → Con → Set where
  [] : Γ ⊢C []
  _::_ : Γ ⊢ A → Γ ⊢C Δ → Γ ⊢C (A :: Δ)
```

The idea is that _⊢C_ corresponds to parallel substitutions. That is an element Γ ⊢C A_0 , A_1 , .. , An is a sequence of derivations a_0 :: a_1 :: .. :: an with Γ ⊢ ai : Ai. We have an identity substitution:

```
idC : Γ ⊢C Γ
```

and we can *cut* a term and a substitution:

```
cut : Γ ⊢C Δ → Δ ⊢ A → Γ ⊢ A
```

Iterating this operation provides us with a cut for _⊢C_ corresponding to the transitivity of the future relation:

```
cutC : Γ ⊢C Δ → Δ ⊢C Θ → Γ ⊢C Θ
```

The forcing relation for our universal model is just derivability applied to variables and monotonicity is just an instance of cut. This completes the construction of the universal Kripke model - the details are again left as an exercise:

```
UM : Kripke
UM = record
  {Worlds = Con
  ; _⊑_ = _⊢C_
  ; refl⊑ = idC
  ; trans⊑ = cutC
  ; _⊩_ = λ Γ x → Γ ⊢ Var x
  ; mon⊩ = λ γ a → cut γ a
  }
```

The derived forcing relation _⊩F_for this model has the same type as the derivation relation. Hence the key lemma is to establish that forcing and derivability coincide. This can be shown by mutual induction over the formula A:

```
q : Γ ⊩F A → Γ ⊢ A
u : Γ ⊢ A → Γ ⊩F A
```

I use q for *quote* sind it goes from the semantics into the syntax, and u for *unquote*. Using these lemmas it is not hard to derive completeness:

```
compl : Γ ⊢ A → Γ ⊨ A
```

8.6 Normalisation

There is an alternative way to establish non-provability which corresponds more to our naive reasoning. We only consider derivations which are normal, which means that they avoid derivations which can be β-reduced, i.e. that correspond to λ-terms of the form (λ x → M) N and which always use λ to prove an implication. The latter corresponds to always η-expanding and is called the long η normal form.

Now when we try to prove the Pierce formula

```
p : ((P → Q) → P) → P
p = ?
```

the first step has to be a λ:

```
p = λ x → ?
```

with x : (P → Q) → P. Now our goal is P and the only assumption that produces something in P is x. Hence we have to refine by x:

```
p = λ x → x ?
```

Ok our goal is now to prove P → Q. Since this is an implication we need to use a λ:

```
p = λ x → x (λ y → ?)
```

We now have a new assumption y : P and our goal is Q. But now we are stuck because none of our assumptions produces Qs. Hence there is no normal proof.

It is clear that to use this idea to show that there is no derivation of ((P → Q) → P) → P in the original system we need to prove that for any derivation Γ ⊢ A there is a normal derivation Γ ⊢nf A.

Let's start with defining normal derivations inductively. First of all we need to recover variables, which correspond to using an assumption from the context.

```
data _⊢v_ : Con → Form → Set where
  zero : (A :: Γ) ⊢v A
  suc : Γ ⊢v A → (B :: Γ) ⊢v A
```

This derivation correspond to numbers, indeed they are de Bruijn indices which we have discussed earlier.

We now mutually define three kinds of derivations:

```
data _⊢nf_ : Con → Form → Set
data _⊢ne_ : Con → Form → Set
data _⊢sp_,_ : Con → Form → Form → Set
```

Here _⊢nf_ are the normal derivations, _⊢ne_ are *neutral derivations* this is they are derivations which are a variable applied to a sequence of normal terms, and _⊢sp_,_ is the *spine* of a neutral derivation, i.e. the sequence of normal terms to which the variable is applies. The spine derivation has two formula parameters: the first one is the assumption we start with and the 2nd one is the goal.

So let's look at _⊢sp_,_ first:

```
data _⊢sp_,_ where
  [] : Γ ⊢sp C , C
  _::_ : Γ ⊢nf A → Γ ⊢sp B , C → Γ ⊢sp (A [⇒] B) , C
```

Once we have reached the end of the sequence the goal has to agree with the premise. The _::_ constructor shortens the distance from the premise A [⇒] B to B by providing Γ ⊢nf A.

The neutral derivations are simply given by a variable applied to a spine:

```
data _⊢ne_ where
  ne : Γ ⊢v A → Γ ⊢sp A , B → Γ ⊢ne B
```

Finally we can define normal derivations:

```
data _⊢nf_ where
  ne→nf : Γ ⊢ne Var x → Γ ⊢nf (Var x)
  lam : (A :: Γ) ⊢nf B → Γ ⊢nf (A [⇒] B)
```

A normal derivation of an implication is always given by lam but when our goal is a variable we have to resort to neutral derivations.

We can now show that there is no normal derivation of the Pierce formula just by using pattern matching:

```
no-Pierce-nf-deriv :
  ¬ ([] ⊢nf (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
```

But as usual I leave this as an exercise.

Now let's prove normalisation. Actually we have already done it more or less in the previous section when we proved completeness. The basic idea is to combine soundness and completeness. That is using soundness we go from a derivation to the semantics in the universal Kripke model. Using completeness we can go back to a derivation but this wouldn't tell us anything new. However, we can observe that the completeness prove always returns a normal derivation! Hence by combining soundness and completeness we can prove normalisation.

The Kripke model we are using is just a variation of the universal model from the last section. The future relation just uses variables since normal derivations are not closed under general substitutions but only under renamings

```
data _⊢vC_ : Con → Con → Set where
  [] : Γ ⊢vC []
  _::_ : Γ ⊢v A → Γ ⊢vC Δ → Γ ⊢vC (A :: Δ)
```

We need to show that we can establish identity and cut operations for variable substitutions:

```
idvC : Γ ⊢vC Γ
cutvC : Γ ⊢vC Δ → Δ ⊢vC Θ → Γ ⊢vC Θ
cutne : Γ ⊢vC Δ → Δ ⊢ne A → Γ ⊢ne A
```

the proofs are the same as before but now applied to the refined derivation forms. Hence we get another Kripke model:

```
NM : Kripke
NM = record
  {Worlds = Con
  ; _⊑_ = _⊢vC_
  ; refl⊑ = idvC
  ; trans⊑ = cutvC
  ; _⊩_ = λ Γ x → Γ ⊢ne Var x
  ; mon⊩ = λ γ a → cutne γ a
  }
```

We also need to refine the types of q and u:

```
q : Γ ⊩F A → Γ ⊢nf A
u : Γ ⊢ne A → Γ ⊩F A
```

And using q and soundness (⟦_⟧) we can prove normalisation:

```
norm : Γ ⊢ A → Γ ⊢nf A
```

Putting norm and no-Pierce-nf-deriv together we derive an alternative proof of the non-derivability of the Pierce formula"

```
no-Pierce-deriv : ¬ ([] ⊢ (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
no-Pierce-deriv p = no-Pierce-nf-deriv (norm p)
```

So in a way normalisation is a canned completeness proof where the semantic component is replaced by a purely syntactic definition of normal forms.

8.7 History

Natural deduction was introduced by Gentzen in the 1930ies [Gen35, Gen35]. In his presentation he didn't use explicit contexts as we did but marked assumptions in derivations and linked them to the position where they are discharged. He also presented the system for predicate logic and covered all logical connectives. Gentzen also introduced a different calculus, the sequent calculus, as a tool to prove his *Hauptsatz* which corresponds to the normalisation theorem we have discussed.

We have already discussed the history of combinatory logic in section 2.6. The completeness proof was first sketched by Schönfinkel [Sch24] it can be found in [Cur30] and in the form we have presented it was first given by Rosenbloom [Ros50] (according to [CH06]).

Kripke modes were first introduced for modal logic [Kri59]. The application to intuitionistic logic is discussed by Schütte in [Sch68]. The completeness theorem in the form we have presented is is only provable for the fragment including only $\to$, $\wedge$, True and $\forall$. To include the remaining connectives, i.e. False, $\vee$, $\exists$ one has to use Beth models [Bet48] (as described in [TvU99]) or resort to a classical metatheory. In general one finds a good overview over this kind of material in the books by Troelstra and van Dalen [TvD88b, TvD88a].

My presentation of normalisation is inspired by *Normalisation by Evaluation* which was discovered by Schwichtenberg and Berger [BS91]. The relation between this algorithm and the completeness proof for Kripke models was noted in [AHS95].

8.8 Exercises

1. Implement the efficient algorithm for translating λ-terms into combinatory logic which was described in section 2.4. That is

   - If the variable x does not appear in M then λ x → M is just K M.
   - λ x → g x is just g.

   You need to reimplement:

   ```
   lam-sk-x : (A :: Γ) ⊢skC B → Γ ⊢skC (A [⇒] B)
   ```

   The rest stays basically the same — we only replace one direction of thm:

   ```
   ⊢→⊢skC-x : Γ ⊢ A → Γ ⊢skC A
   ⊢→⊢skC-x zero          = zero
   ⊢→⊢skC-x (suc d)       = suc (⊢→⊢skC-x d)
   ⊢→⊢skC-x (lam d)       = lam-sk-x (⊢→⊢skC-x d)
   ⊢→⊢skC-x (app d e)  = app (⊢→⊢skC-x d) (⊢→⊢skC-x e)
   thm-x : ([] ⊢ A) → ⊢sk A
   thm-x d = skC→sk (⊢→⊢skC-x d)
   ```

Try the improved version on

```
d-C-sk-x : ⊢sk (((Var "Q") [⇒] (Var "R")) [⇒] ((Var "P") [⇒] (Var "Q"))
                [⇒] ((Var "P") [⇒] (Var "R")))
d-C-sk-x = thm-x d-C
```

the result should be:

```
app (app SS (app KK SS)) KK
```

2. Complete the soundness proof for Kripke models from section 8.4. That is prove the following propositions:

```
mon-F : w₁ ⊑ w₀ → {A : Form} → w₀ ⊩F A → w₁ ⊩F A
mon-C : w₁ ⊑ w₀ → {Γ : Con} → w₀ ⊩C Γ → w₁ ⊩C Γ
⟦_⟧ : Γ ⊢ A → Γ ⊨ A
```

3. Complete the proof that the Pierce formula is not derivable using the soundness result and by completing the definition of the Kripke countermodel.

   Complete the definition of the Kripke model KP:

```
trans⊑P : {w₀ w₁ w₂ : W} → w₀ ⊑P w₁ → w₁ ⊑P w₂ → w₀ ⊑P w₂
mon⊩P : {w₀ w₁ : W}
  → w₁ ⊑P w₀ → {x : String} → w₀ ⊩P x → w₁ ⊩P x
KP : Kripke
KP = record
  {Worlds = W
  ; _⊑_ = _⊑P_
  ; refl⊑ = refl
  ; trans⊑ = trans⊑P
  ; _⊩_ = _⊩P_
  ; mon⊩ = mon⊩P
  }
```

   And then using KP show that Pierce is not derivable:

```
not⊩Pierce : ¬ (w0 ⊩F (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
no-Pierce-deriv : ¬ ([] ⊢ (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
```

4. Complete the completeness proof from section 8.5, that is complete the definition of the universal Kripke model:

```
idC : Γ ⊢C Γ
cut : Γ ⊢C Δ → Δ ⊢ A → Γ ⊢ A
```

```
cutC : Γ ⊢C Δ → Δ ⊢C Θ → Γ ⊢C Θ
UM : Kripke
UM = record
  {Worlds = Con
  ; _⊑_ = _⊢C_
  ; refl⊑ = idC
  ; trans⊑ = cutC
  ; _⊩_ = λ Γ x → Γ ⊢ Var x
  ; mon⊩ = λ γ a → cut γ a
  }
```

Hint: It is useful to establish the following lemmas generalising weakening:

```
sucC : Γ ⊢C Δ → (B :: Γ) ⊢C Δ
extC : Γ ⊢C Δ → (A :: Γ) ⊢C (A :: Δ)


q : Γ ⊩F A → Γ ⊢ A
u : Γ ⊢ A → Γ ⊩F A
qC : Γ ⊩C Δ → Γ ⊢C Δ
compl : Γ ⊢ A → Γ ⊨ A
```

5. Show that there is no normal derivation of the Pierce formula:

```
no-Pierce-nf-deriv : ¬ ([] ⊢nf (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
```

6. Complete the construction of the Kripke model using normal forms as described in section 8.6. That is complete the following definitions by adapting the construction of the previous question:

```
idvC : Γ ⊢vC Γ
cutvC : Γ ⊢vC Δ → Δ ⊢vC Θ → Γ ⊢vC Θ
cutne : Γ ⊢vC Δ → Δ ⊢ne A → Γ ⊢ne A
NM : Kripke
NM = record
  {Worlds = Con
  ; _⊑_ = _⊢vC_
  ; refl⊑ = idvC
  ; trans⊑ = cutvC
  ; _⊩_ = λ Γ x → Γ ⊢ne Var x
  ; mon⊩ = λ γ a → cutne γ a
  }
```

Once this is done we can derive the variations of q and u:

```
q : Γ ⊩F A → Γ ⊢nf A
u : Γ ⊢ne A → Γ ⊩F A
```

Finally use this to derive the normalisation function:

```
norm : Γ ⊢ A → Γ ⊢nf A
```

which can be use to provide an alternative proof of the non derivability of the Pierce formula:

```
open NormPierce
no-Pierce-deriv : ¬ ([] ⊢ (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
no-Pierce-deriv p = no-Pierce-nf-deriv (norm p)
```