Chapter 5

Introduction to Curry-Howard isomorphism for classical Logic

5.1 Game-style semantics

In order to explain the computational content of classical proofs, it is useful to first go back to the intuitionistic case. It is possible to explain Heyting’s semantics with a framework of game semantics.

In this setting, a proposition is viewed as a game between the mathematician, who aims to prove the proposition, and an opponent (sometimes called the nature). Depending of the form of the proposition, it is either the mathematician’s turn, or the nature’s turn to play.

- If the proposition is $\exists x. A$, the mathematician has to provide a witness $t$. The game goes on with $A[x \backslash t]$.

- If the proposition is $\forall x. A$, the nature provides a possible counter-example $t$ and the game goes on with $A[x \backslash t]$.

- If the proposition is atomic and closed, then if the proposition is true, the mathematician wins; if the proposition is false, nature wins. In particular, if the proposition is $\bot$, nature wins.

- If the proposition is $A \lor B$, nature choses "left or right" and the game goes on with either $A$ or $B$.

- If the proposition is $A \land B$, the mathematician has to win the game first for $A$ and then for $B$.

The case of $A \Rightarrow B$ is a little more delicate de treat precisely. For the present matter, it is sufficient to think of it as an abbreviation of $B \lor \neg A$.

Because of the obvious duality in the way the two quantifiers are treated, one sometimes refers to the two players as “Abélard” (of $\forall$abélard) and “Éloïse” (or $\exists$loïse).

The important point though is that in this setting, an intuitionistic proof of proposition $A$ corresponds to a winning strategy in the game corresponding to $A$.

What we will see is that a classical proof of $A$ corresponds to a winning strategy in the game where the rules have been slightly changed in favor of the mathematician: in the classical game, the mathematician is allowed to take back some moves; he can sometimes say “oh, I made a mistake; let us go back a a previous situation”. This will be made possible expressing the proof (or strategy) not anymore in a purely functional language (as is the case for intuitionistic proofs) but a language extended by control operators like call/cc.
5.2 Double Negation translations

An older approach to study the link between classical and intuitionistic proofs goes back to the 1930ties. It is based on the fact that if \( A \lor \neg A \) is not intuitionistically provable in general, \( \neg\neg(A \lor \neg A) \) is. The idea of double negation translations is thus simply to add double negations at crucial stages in the proposition to be proved. Thus, instead of classically proving a proposition \( A \), one intuitionistically proves a (weaker) proposition \( A^* \). To be precise, we will see that \( A^* \) can even be proved in minimal logic.

There are many such translations, that is many ways to insert the double negations. We here chose one.

**Definition 5.2.1** Let \( A \) be a proposition in a first-order language. We define the proposition \( \neg\neg A \) in the same language by the following equations:

\[
\begin{align*}
A \Rightarrow B &= \neg A \Rightarrow \neg B \\
\forall x.A &= \forall x.\neg A \\
\exists x.A &= \exists x.\neg A \\
A \lor B &= (\neg A) \lor (\neg B) \\
A \land B &= (\neg A) \land (\neg B)
\end{align*}
\]

Finally, when \( \Gamma \) is a set of assumptions \( A_1, \ldots, A_n \), we define \( \neg\neg \Gamma \) as the set \( A_1, \ldots, \neg A_n, \ldots \).

**Remark 5.2.1** By noticing that \( A \Rightarrow \neg\neg A \), it is easy to check that \( A \Rightarrow \neg\neg A \) holds in minimal logic for any proposition \( A \).

The other point is that the double negation translation of the excluded middle (or other classical schemes) is intuitionistically provable.

**Lemma 5.2.1** For any propositions \( A \) and \( B \), the following propositions are provable in minimal logic:

\[\neg\neg(A \lor \neg A), \neg\neg(\neg A \Rightarrow A), \neg\neg((A \Rightarrow B) \Rightarrow A)\].

**Lemma 5.2.2** For any proposition \( A \), there exists a derivation of \( \neg\neg(\bot) \Rightarrow \neg\neg A \) in minimal logic.

**Proof 8** We have \( \neg\neg(\bot) \Rightarrow \neg\neg A = \neg\neg(\bot \Rightarrow \neg\neg A) \). Consider:

\[
\begin{align*}
\neg(\bot \Rightarrow \neg\neg A) ; \bot \Rightarrow \neg\neg A & \vdash (Ax) \\
\neg(\bot \Rightarrow \neg\neg A) ; \bot \Rightarrow \neg\neg A & \vdash (\Rightarrow-E)
\end{align*}
\]

**Theorem 5.2.1** If there exists a classical derivation of \( \Gamma \vdash A \), then there exists an intuitionistic derivation of \( \Gamma \vdash \neg\neg A \).

**Proof 9** The proof is, of course, by induction over the classical derivation of \( \Gamma \vdash A \). The lemmas above already treat the cases of excluded middle, and of the rule \( \bot \Rightarrow \).

We do not detail the cases of all the rules. Instead of writing trees in natural deduction, it is more compact to write the corresponding typed \( \lambda \)-terms. Given \( \Gamma \vdash t : A \), we construct a term \( \overline{\Gamma} \) such that \( \overline{\Gamma} \vdash \overline{t : \neg\neg A} \).

(Axiom) This corresponds to the case of the variable. We have \( (x : A) \in \overline{\Gamma} \) and need to construct a term of type \( \neg\neg A \). We do so by taking \( \overline{x} = \lambda k : \neg A. (k x) \).
5.2. DOUBLE NEGATION TRANSLATIONS

(implies-\e) We have $\Gamma \vdash t : A \rightarrow B$ and $\Gamma \vdash u : A$ and thus

$$\Gamma \vdash \lambda t : \neg \neg (A \rightarrow \neg \neg B)$$

and $\Gamma \vdash u : \neg \neg A$.

We need to construct $(t \ u) : \neg \neg B$. An interesting point is that we here have the choice between two similar definitions. One puts $t$ first, the other one $u$:

$$(t \ u) = \lambda k : \neg B.(t \ \lambda f : A \rightarrow \neg \neg B.(u \ \lambda a : A.(f \ a \ k)))$$
or

$$(t \ u) = \lambda k : \neg B.(u \ \lambda a : A.(t \ \lambda f : A \rightarrow \neg \neg B.(f \ a \ k))).$$

($\Rightarrow$-1) We have $\Gamma(x : A) \vdash t : B$ and thus $\Gamma(x : A) \vdash t : \neg \neg B$. We need to construct a term of type $\neg \neg (\neg \neg A \rightarrow \neg \neg B)$. We can take:

$$\lambda x : A.t = \lambda k : \neg (\neg \neg A \rightarrow \neg \neg B).(k \ \lambda x : A.t).$$

($\land$-1) We mention this case because it is another nice example where we have to break the symmetry. We have $\Gamma \vdash t : A$ and $\Gamma \vdash u : B$; thus also $\Gamma \vdash t : \neg \neg A$ and $\Gamma \vdash u : \neg \neg B$.

We can define :

$$(t, u) = \lambda k : \neg (\neg \neg A \land \neg \neg B).(t \ \lambda a : A.(u \ \lambda b : B.(k \ (a, b))))$$

but also put $u$ first with another possible definition:

$$(t, u) = \lambda k : \neg (\neg \neg A \land \neg \neg B).(u \ \lambda b : B.(t \ \lambda a : A.(k \ (a, b))))).$$

Getting rid of the double negations

In the general case, this is as far as we can go. It has been noticed however, that in some cases, one can get rid of the double negations, that is go back from a proof of $\neg \neg A$ to an intuitionistic proof of the original proposition $A$. The following steps are generally attributed to Harvey Friedman.

The quantifier-free case

We consider the case of theories were all atomic propositions are decidable. This includes, in particular, arithmetic, since $\forall x, \forall y, x \equiv y \lor x \neq y$ is provable in Heyting's arithmetic.

Lemma 5.2.3 If all atomic propositions are decidable, then for any quantifier-free proposition $A$ the proposition $A \lor \neg A$ is provable intuitionistically.

Proof 10 Quite easy by induction over the structure of $A$. For instance, suppose we have $B \lor \neg B$ and $C \lor \neg C$, then one proves the decidability of $B \land C$ by cases:

- if $B$ and $C$ are true, then so is $B \land C$,
- if $\neg B$ holds, or $\neg C$ holds, then $\neg (B \land C)$ holds.

The cases of $B \lor C$ and $B \Rightarrow C$ are similar. The case of $\bot$ is trivial ($\bot \Rightarrow$ holds).

In a similar way, we can prove:

Lemma 5.2.4 If all atomic propositions are decidable, then for any quantifier-free proposition $A$, when can show intuitionistically $A \iff \neg \neg A$.

Now we can use Friedman's trick. When we have a derivation of $\Gamma \vdash \neg \neg A$ in minimal logic, because the rule $\bot$-\e is not used, we can replace $\bot$ by any arbitrary proposition in the derivation. Friedman's idea is then to replace $\bot$ by the proposition $A$ itself.
CHAPTER 5. INTRODUCTION TO CURRY-HOWARD ISOMORPHISM FOR CLASSICAL LOGIC

Theorem 5.2.2 If \( A \) is a quantifier-free proposition, and \( \exists x.A \) is provable in classical arithmetic, then \( \exists x.A \) is also provable in Heyting’s arithmetic.

Proof 11 By double negation translation, we obtain a proof in minimal logic of \( \neg \neg \exists x.A \), which by the lemma above, gives us a proof of \( \neg \neg \exists x.A \) in minimal logic, that is of \( ((\exists x.A) \Rightarrow \bot) \Rightarrow \bot \).

We can replace in the whole derivation every occurrence of \( \bot \) by \( \exists x.A \). This gives us, still in minimal logic, a derivation of:

\[
( (\exists x.A) \Rightarrow \exists x.A ) \Rightarrow \exists x.A.
\]

Since \( (\exists x.A) \Rightarrow \exists x.A \) is obvious, we obtain an intuitionistic proof of \( \exists x.A \).

To sum up, it is possible, using double-negations, to transform a classical proof into a proof with computational content. Until the early 1990ties, this computational content was still mysterious: applying double negation to non-trivial proofs, generally yields very complicated and intricate programs.

5.3 control operators

Control operators in functional languages allow the programmer to take advantage of the evaluation order. Let us use a presentation as simple as possible. We consider only \( \lambda \)-calculus, that is no pairs or sum-types, although they can be included in what follows.

5.3.1 Global reduction rules

The reduction rules of control operators are global. In order to capture this, we use a presentation of terms with evaluation contexts. Like in programming languages, we consider weak reductions, that is reductions that do not occur under \( \lambda \)-abstraction. This means the evaluation context of the reduced redex is made only of applications.

\[
C ::= [] | (tC) | (Ct)
\]

\[
t ::= x | x.t | (tt) | C \mid A(t)
\]

The regular weak \( \beta \)-reduction is thus:

\[
C[(\lambda x.t \ u)] \Rightarrow C[t[x \ u]].
\]

The point of the “abort” \( A \) operator is just it can exit from its evaluation context up to the toplevel:

\[
C[A(t)] \Rightarrow t.
\]

The point of the \( C \) operator, is that it can capture its evaluation context. Its reduction rule is:

\[
C[C(t)] \Rightarrow (t \ \lambda x.A[C[x]]).
\]

In other words, \( C(t) \) leaves its evaluation context, but has the possibility the return to it through the closure \( \lambda x.A(t) \).

5.3.2 Typing

The link with classical logic becomes obvious when typing the operators. Let us call \( A_T \) the type of the “toplevel”, that is the type of \( C[] \). Let us call \( B \) the type of \( [] \) in the context \( C \).

Since \( A(t) \) involves that \( t \) becomes the toplevel, we require \( t : A_T \). On the other hand, \( A(t) \) can be considered of any type. Formally:

\[
t : A_T \\
A(t) : B
\]
5.3. CONTROL OPERATORS

Logically, it means that \( \mathcal{A} \) corresponds to the proposition \( A_T \Rightarrow \bot \). This means also that it corresponds to a form of proof by contradiction: in order to prove \( A_T \) we can assume \( \neg A_T \).

Looking at \( \mathcal{C}(t) \), it can be of any type, provided that \( t \lambda x.\mathcal{A}(x) \) is of type \( \mathcal{A} \). We see that \( x \) needs to be of type \( B \). Which means that \( \lambda x.\mathcal{A}(\mathcal{C}[x]) \) is of type \( \neg B \); thus \( t : (\neg B) \rightarrow A_T \) and \( \mathcal{C}(t) : B \).

5.3.3 Control operator with local rules

To show in a simpler way how control operators behave, we can replace \( \mathcal{C} \) by a binary operator \( \mathcal{B}(t, x.u) \), which binds the variable \( x \) in its second argument \( u \). The typing shows how this operator allows to eliminate double negation:

\[
\Gamma \vdash t : (B \rightarrow A_T) \rightarrow A_T \quad \Gamma(x : B) \vdash u : C
\]

\[
\Gamma \vdash \mathcal{B}(t, x.u) : B
\]

The reduction rules show how this operator progressively captures its environment:

\[
(B(t, x.u) \ v) \triangleright_B B(t, x.(u \ v))
\]

\[
(v \ B(t, x.u)) \triangleright_B B(t, x.(v \ u))
\]

Furthermore, the following rule can only be executed at toplevel:

\[
B(t, x.u) \triangleright_B (t \ \lambda x.\mathcal{A}(u)).
\]

It is easy to see how the typing corresponds to classical logic, and that these reduction rules preserve typing.

5.3.4 The question of confluence

There an obvious problem however: as given above, the reduction rules are not confluent. The critical pairs are:

- \( (B(t_1, x_1.u_1) \ B(t_2, x_2.u_2)) \) which can reduce to \( B(t_1, x_1.(u_1 \ B(t_2, x_2.u_2))) \) and also reduce to \( B(t_2, x_2.\ (B(t_1, x_1.u_1) \ u_2)) \),

- \( (\lambda x_1.\ t_1 \ B(t_2, x_2.u)) \) which can reduce to \( t_1[x_1 \ B(t_2, x_2.u)] \) and to \( B(t_2, x_2.(\lambda x_1.\ t_1 \ u)). \)

Furthermore, there will be even more such critical pairs when the language is extended with regular constructs like pairs.

This means that in practice, one will want to fix the evaluation order, in a language with control operators. This can be done even when using reduction rules by using the notion of values.

**Definition 5.3.1** A value is a term which is of the form \( \lambda x.t \) or of the form \( x \).

Remark that a value cannot reduce to a term \( \mathcal{B}(t, x.u) \).

One then obtains a confluent set of rules by requesting that some of the terms involved are values. The following is the set of rules commonly known as call-by-value reduction; the point is that \( v \) denotes a value.

\[
(\lambda x.t \ v) \triangleright t[x \ \backslash \ v]
\]

\[
(B(t, x.u) \ v) \triangleright B(t, x.(u \ v))
\]

\[
(v \ B(t, x.u)) \triangleright B(t, x.(v \ u))
\]
The following set corresponds to call-by-name semantics:

\[(\lambda x.t\ v) \triangleright t[x\ \hat{\\\\quad} v] \quad (B(t,\ x.u)\ v) \triangleright B(t,\ x.(u\ v))\]

And the following set to a variant of call-by-value with right-to-left evaluation of the arguments:

\[(\lambda x.t\ v) \triangleright t[x\ \hat{\\\\quad} v] \quad (B(t,\ x.u)\ v) \triangleright B(t,\ x.(u\ v)) \quad (v\ B(t,\ x.u)) \triangleright B(t,\ x.(v\ u))\]

We do not study this here, but the fact that one classical proof of \(\exists x.A\) can yield several possible witnesses, and thus that a confluent strategy must arbitrarily “chose” one between them is something which had been known long ago (the so-called “cross-cut” elimination in sequent calculus).

5.3.5 CPS-transformation and termination

It is possible to transform a program with control operators into a purely functional program which behaves the same way. The principle of such a transformation is that each node in the program is translated into a function which takes an additional argument corresponding to its continuation, which is basically what remains to be done with the value of the node. The program transformations are called continuation-passing-style translations, or CPS. Different CPS correspond to the different reduction styles (CBN, CBV).

In 1990, Tim Griffin noticed that these CPS when applied to typed programs, actually corresponded to double-negation translations, which explains how they can transform a program with control operators (corresponding to classical logic) to purely functional programs (corresponding to intuitionistic logic).

In other words, taking a program \(t : A_T\) with control operators, we translate it into a purely functional program \(\overline{t}\) through the transformation used in the proof of theorem 5.2.1. We need to extend this transformation for the control operators; for instance by:

\[\overline{A(t)} = \lambda k : T \rightarrow A_T.(\overline{\lambda x : A_T.x})\]

which means that \(A(t)\) throws away its continuation and replaces it by the “empty” one.

In the same vein, suppose \(t : (B \rightarrow A_T) \rightarrow A_T\) and \((x : B) \vdash u : C\). We can take:

\[\overline{B(t,\ x.u)} = \lambda k : T \rightarrow A_T.(\overline{\lambda x : B.x})\lambda k' : (A_T \rightarrow A_T).((\pi k)\ \lambda k' : A_T.k').\]

It is possible to see that the reductions in the program with control operators can be emulated by regular \(\beta\)-reductions of the CPS-translated program. We only sketch the obviously tedious proofs.

Remark that when \(t : B\) is a value \((x\ or\ \lambda x.u)\), then \(\overline{t}\) is of the form \(\lambda k : \overline{B} \rightarrow A_T.(k\ t')\) with \(k\) not free in \(t'\) and \(t' : \overline{B}\).

Using this, one can remark that \(\overline{(\lambda x.t\ v)} \triangleright_{\beta} \overline{t[x\ \hat{\\\\quad} v]}\) and \(\overline{t[x\ \hat{\\\\quad} v]} = \overline{t[x\ \hat{\\\\quad} \overline{v}]}\).

Similarly, when \(t \triangleright_{\beta} t', \overline{t'} \triangleright_{\beta} \overline{\overline{t'}}\).

In other words, once a reduction strategy is fixed and corresponds to a CPS-translation, one can show strong normalization for \(\triangleright_{\beta\beta}\).
5.4. WHAT ABOUT DEPENDENT TYPES?

5.3.6 When no operators lurk around
This allows to retrieve Friedman’s result: if $t : A$ and $A$ contains no $\forall$ and no $\Rightarrow$, then the normal form of $t$ contains no continuation operator anymore. To prove this precisely, one needs to detail the CPS-translation for all the cases, which is something we do not do here; but there is no fundamental difficulty.

5.4 What about dependent types?
The typing of control operators is not incompatible with dependent types, but there is one important restriction: control operators should not appear inside types. This is not surprising since it is difficult to make sense from propositions like "this control operator is an even number".

Building a system with dependent types which prevents certain terms (here control operators) from appearing inside types is not very difficult, but not detailed here.

5.5 Going further
The discovery of the link between control operators and classical logic launched a new research direction in order to define new programming languages which capture in cleaner ways to computational content of classical proofs. Among these, we mention Parigot’s $\lambda\mu$-calculus, and Herbelin’s calculi which apply the ideas of $\lambda\mu$ to classical sequent calculus.