Lecture 2
From Computability to Program Theory, Part 2
The λ-calculus

Davide Barbarossa

Contents

1 Introducing the λ-terms

Fix now and for all a countable set Var of variables. Consider the set of finite directed graphs M, which we call *nets*[1], such that: the internal nodes of M (i.e. those with at least one child) are in $\{@, \lambda, \bullet\}$; the leaves of M (i.e. those with no children) are in Var; all @-nodes of M have exactly one left child, one right child and one parent; M has exactly one node, called its *conclusion* $r(M)$, with the property that all its parents (if any) are $\bullet$-nodes. The set $FV(M) := \{x \in \text{Var} \mid x \textit{ is a leaf of } M\}$ is called the set of *free variables* of M. For x a variable, we denote by x the net whose only node (thus, its conclusion) is the leaf x. There is nothing special in the use of nets as above: they are just a handy framework in order to perform the following operations:

- An operation @ which, given two nets M, N, returns the net, denoted MN, obtained from the disjoint union of M, N by adding a conclusion node @ and two edges $r(N) \leftarrow\ @ \rightarrow\ r(M)$.
- For each variable y, an operation λy which, given a net M returns the net, denoted by $\lambda y.M$, obtained from M by adding a new conclusion node λ, an edge $\lambda \rightarrow r(M)$, and by replacing all the leaves y of M (if any) by $\bullet$ and adding an edge $\bullet \rightarrow \lambda$ for each of those.

Remark 1.1. *Given a net M, define $M\{x := y\}$ to be the net obtained from M by replacing all occurrences of x in M (necessarily leaves in M) with y. Then $M\{x := y\}$ is still a net and:*

$$FV(M\{x := y\}) = \begin{cases} (FV(M) - \{x\}) \cup \{y\} & \textit{if } x \in FV(M) \\ FV(M) & \textit{otherwise.} \end{cases}$$

[1]This is neither a standard terminology nor a standard notion.

Remark that $(\lambda z.M)\{x := y\} = \lambda z.(M\{x := y\})$ if $z \neq x, y$, but if $z \in \{x, y\}$ the equality may fail[2]*. Moreover, immediately from the definition of nets, we have e.g. $\lambda x.x = \lambda y.y$. In general:*

$$\lambda x.M = \lambda y.(M\{x := y\}) \quad \textit{whenever } y \notin FV(M). \tag{α}$$

The fact that all subnets of a net which are λ-abstractions can be rewritten in this way, is called α-conversion*. It says that (typically during an induction), we can always choose the variable popped out from the deconstruction of a λ-abstraction as "fresh", i.e.:* for each finite collection of nets, we can assume that the abstracted variable does not occur (free) in any of them*. In our particular framework of nets as above, α-conversion holds for free.*

Definition 1.2. *A* λ-term-with-context-variables *is a pair of a finite set of variables $\underline{\mathtt{x}}$, called* context variables*, and a net $\mathtt{M}$, and which is defined inductively by*[3]*:*

$$(\mathtt{x} \in \underline{\mathtt{x}})\frac{}{\underline{\mathtt{x}} \vdash \mathtt{x}}(\mathit{proj}^{\underline{\mathtt{x}}}_{\mathtt{x}}) \qquad (\mathtt{y} \notin \underline{\mathtt{x}})\frac{\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{M}}{\underline{\mathtt{x}} \vdash \lambda\mathtt{y}.\,\mathtt{M}}(\lambda\mathtt{y}) \qquad \frac{\underline{\mathtt{x}} \vdash \mathtt{M} \qquad \underline{\mathtt{x}} \vdash \mathtt{N}}{\underline{\mathtt{x}} \vdash \mathtt{M\,N}}(@)$$

If $\underline{\mathtt{x}} \vdash \mathtt{M}$, we say that the net $\mathtt{M}$ admits context variables (or that $\underline{\mathtt{x}}$ is adequate for $\mathtt{M}$).

The idea is that a λ-term-with-context-variables $\underline{\mathtt{x}} \vdash \mathtt{M}$ represents a program with input points of R bound to the context variables $\underline{\mathtt{x}}$, and output the point of R represented by $\mathtt{M}$. The rules above represent then the kind of manipulations of functions from Theorem 4.9 of Lecture 1.

Note that a λ-term-with-context-variables can have different derivations, e.g. $\dfrac{\mathtt{x} \vdash \mathtt{x}}{\vdash \lambda\mathtt{z.z}}$ and $\dfrac{\mathtt{y} \vdash \mathtt{y}}{\vdash \lambda\mathtt{z.z}}$. The next Lemma 1.3, Proposition 1.4 and Lemma 1.6 are proven in the Section 3.

Lemma 1.3. *The following two rules are admissible, resp. called* σ-renaming *and* Weakening*:*

$$(\sigma : \underline{\mathtt{x}} \overset{\simeq}{\to} \underline{\mathtt{z}} \ \textit{ bijection})\frac{\underline{\mathtt{x}} \vdash \mathtt{M}}{\underline{\mathtt{z}} \vdash \mathtt{M}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}}(\sigma) \qquad\qquad \frac{\underline{\mathtt{x}} \vdash \mathtt{M}}{\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{M}}(W)$$

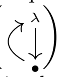
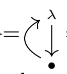
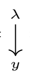
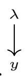
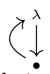
Proposition 1.4. *The nets admitting context variables (possibly an empty list) are exactly the ones obtained by the following inductive rules. They are called* λ-terms*, and we call Λ their set.*

$$\mathtt{M} ::= \mathtt{x} \mid \lambda\mathtt{x}.\,\mathtt{M} \mid \mathtt{M\,M} \tag{Λ}$$

A λ-term which admits the empty list of context variables is said to be closed.

The idea is that λ (resp. in the sense of R, of rule for λ-term-with-context-variables and of λ-abstraction) encodes a function in a sense external to the respective framework under consideration as a function in a sense internal to it. In λ-calculus, those functions are "intentional", and we are going to see that the framework is a programming language.

Example 1.5. $\mathtt{I} := \lambda\mathtt{x.x} \in \Lambda$ *is closed. Of course, it still admits many other context variables, e.g.* $\dfrac{\mathtt{x, y, z} \vdash \mathtt{z}}{\mathtt{x, y} \vdash \mathtt{I}}$*. Other notable closed λ-terms are in Figure 1. Prove that those nets are indeed λ-terms by using the rules (Λ).*

[2]For a counterexample take, in both cases, $x \neq y$ and $M := x$. Indeed:
$(\lambda x.x)\{x := y\} = \Big($  $\Big)\{x := y\} =$  $\neq$  $= \lambda x.(x\{x := y\})$ and $(\lambda y.x)\{x := y\} =$  $\neq$  $= \lambda y.(x\{x := y\})$.

[3]The comma "," in the contexts denotes union of sets; the condition at the left of a rule in parenthesis is the condition required in order for the rule to be applicable; the sign at the right of a rule is simply its name.

Lemma 1.6. *The free variables of λ-terms satisfy the following recursive equations:*

$$FV(\mathtt{x}) = \{\mathtt{x}\}, \quad FV(\mathtt{M\,N}) = FV(\mathtt{M}) \cup FV(\mathtt{N}), \quad FV(\lambda\mathtt{x}.\,\mathtt{M}) = FV(\mathtt{M}) - \{\mathtt{x}\}.$$

Moreover, $FV(\mathtt{M})$ is contained in any context variable list for $\mathtt{M}$, and $FV(\mathtt{M}) \vdash \mathtt{M}$.

Remark 1.7. *One usually defines λ-terms in a more syntactic way, and then α-conversion would need to be properly defined as an equivalence, and work in the quotient. Here, (α) holds already for nets. In practice, the two approaches are the same, though: every time that we perform an induction, in either approach we need to handle this situation by hand by invoking (α)!*

2 The λ-calculus as a programming language

2.1 Introduction to Denotational Semantics (in R)

We extracted λ-terms precisely from the crucial operations in R, so we expect them to be naturally related with elements of R. Indeed they are, and this is how:

Definition 2.1. *We define, by induction on $\underline{\mathtt{x}} \vdash \mathtt{M}$, its R-*interpretation *$[\![\underline{\mathtt{x}} \vdash \mathtt{M}]\!] : R^{\underline{\mathtt{x}}} \to R$ as:*

$$\begin{array}{llcl}
[\![\underline{\mathtt{x}} \vdash \mathtt{x}]\!] := \mathrm{proj}^{\underline{\mathtt{x}}}_{\mathtt{x}} : R^{\underline{\mathtt{x}}} \to R, \quad \textit{i.e.} & [\![\underline{\mathtt{x}} := \underline{a} \vdash \mathtt{x}]\!] & := & a_{\mathtt{x}} \\
[\![\underline{\mathtt{x}} \vdash \lambda\mathtt{y}.\mathtt{P}]\!] : R^{\underline{\mathtt{x}}} \to R, & [\![\underline{\mathtt{x}} := \underline{a} \vdash \lambda\mathtt{y}.\mathtt{P}]\!] & := & \lambda\left([\![\underline{\mathtt{x}} := \underline{a}, \mathtt{y} := (\_) \vdash \mathtt{P}]\!]\right) \;\; \textit{if } \mathtt{y} \notin \underline{\mathtt{x}} \\
[\![\underline{\mathtt{x}} \vdash \mathtt{P\,Q}]\!] : R^{\underline{\mathtt{x}}} \to R, & [\![\underline{\mathtt{x}} := \underline{a} \vdash \mathtt{P\,Q}]\!] & := & @([\![\underline{\mathtt{x}} := \underline{a} \vdash \mathtt{P}]\!]\;,\;[\![\underline{\mathtt{x}} := \underline{a} \vdash \mathtt{Q}]\!])
\end{array}$$

where we wrote $[\![\underline{\mathtt{x}} := \underline{a} \vdash \mathtt{M}]\!]$ for $[\![\underline{\mathtt{x}} \vdash \mathtt{M}]\!]\,(\underline{a})$. Remark that for a closed $\vdash \mathtt{M}$, we have $[\![\vdash \mathtt{M}]\!] \in R$.

Remark 2.2. *We should justify that the definition above makes sense, in that it defines a unique function. This is both intuitively clear and technically non-trivial! We detail it in Section 3.*

It is evident from the definition that we defined some sort of homomorphism wrt $\lambda, @$, or some sort of functor. We will not precise this, because it would require notions from combinatory algebras or category theory, but the intuition is correct.

Remark 2.3. *Our interpretation map sees $[\![\underline{\mathtt{x}} \vdash \mathtt{M}]\!]$ as a function on $R^{\underline{\mathtt{x}}} \to R$. Another common way, essentially equivalent, is to define the interpretation directly of a term, as a family $[\![\mathtt{M}]\!] : \rho \in R^{\mathrm{Var}} \mapsto [\![\mathtt{M}]\!]_\rho \in R$, indexed by an* environment *ρ, which chooses the value of the variables all at once for* all *terms. The approach taken here is slightly more natural if one thinks of it as a special case of the denotational semantics for typed terms in CCCs or in closed multicategories.*

We know from the previous lecture that equation

$$\mathit{fun} \circ \lambda = \mathrm{id}_{[R \to R]} \tag{β}$$

holds in R. It is interesting to look at its consequence on the interpretation of terms. Namely, given $\underline{\mathtt{y}}, \mathtt{x} \vdash \mathtt{M}$ with $\mathtt{x} \notin \underline{\mathtt{y}}$, and $\underline{\mathtt{y}} \vdash \mathtt{N}$, let us look at $\left[\!\!\left[\underline{\mathtt{y}} \vdash (\lambda\mathtt{x}.\mathtt{M})\,\mathtt{N}\right]\!\!\right]$. The λ-term-with-context-variables of this shape are exactly those where the composition $\mathit{fun} \circ \lambda$ appears. It is immediate (see Section 3) to see that the above interpretation is the function which gives, for each $\underline{a} \in R^{\underline{\mathtt{x}}}$, the element

$$\left[\!\!\left[\underline{\mathtt{y}} := \underline{a} \vdash (\lambda\mathtt{x}.\mathtt{M})\,\mathtt{N}\right]\!\!\right] = \left[\!\!\left[\underline{\mathtt{y}} := \underline{a}, \mathtt{x} := \left[\!\!\left[\underline{\mathtt{y}} := \underline{a} \vdash \mathtt{N}\right]\!\!\right] \vdash \mathtt{M}\right]\!\!\right] \in R$$

That is, it is the application of $\left[\!\left[\underline{\mathtt{y}} := \underline{a}, \mathtt{x} \vdash \mathtt{M}\right]\!\right] : R \to R$ to $\left[\!\left[\underline{\mathtt{y}} := \underline{a} \vdash \mathtt{N}\right]\!\right] \in R$. In other words, it is the substitution of the interpretation of $\mathtt{N}$ for $\mathtt{x}$ in the interpretation of $\mathtt{M}$.

The crucial observation is now that the substitution above can be actually internalised in the language, i.e. *it is itself the interpretation of a λ-term-with-context-variables*:

Definition 2.4. *Given* $\mathtt{M}, \mathtt{N} \in \Lambda$, $\mathtt{x} \in \mathrm{Var}$, *define the term* $\mathtt{M}\{\mathtt{x} := \mathtt{N}\} \in \Lambda$ *by induction*[4]*:*

$$\mathtt{x}\{\mathtt{x} := \mathtt{N}\} := \mathtt{N} \qquad \mathtt{z}\{\mathtt{x} := \mathtt{N}\} := \mathtt{z} \qquad (\mathtt{P}\,\mathtt{Q})\{\mathtt{x} := \mathtt{N}\} := (\mathtt{P}\{\mathtt{x} := \mathtt{N}\})(\mathtt{Q}\{\mathtt{x} := \mathtt{N}\})$$

$$(\lambda \mathtt{z}.\mathtt{P})\{\mathtt{x} := \mathtt{N}\} := \lambda \mathtt{z}.(\mathtt{P}\{\mathtt{x} := \mathtt{N}\}), \quad \textit{if } \mathtt{z} \notin FV(\mathtt{N}) \cup \{\mathtt{x}\}.$$

In terms of nets, the substitution operation is very natural[5]: $\mathtt{M}\{\mathtt{x} := \mathtt{N}\}$ coincides with the net obtained from $\mathtt{M}$ by substituting all (if any) its $\mathtt{x}$-leaves with $\mathtt{N}$.

Theorem 2.5. *For all* $\underline{\mathtt{y}}, \mathtt{x} \vdash \mathtt{M}$ *with* $\mathtt{x} \notin \underline{\mathtt{y}}$, *and* $\underline{\mathtt{y}} \vdash \mathtt{N}$, *we have:*

$$\left[\!\left[\underline{\mathtt{y}} \vdash (\lambda \mathtt{x}.\,\mathtt{M})\,\mathtt{N}\right]\!\right] = \left[\!\left[\underline{\mathtt{y}} \vdash \mathtt{M}\{\mathtt{x} := \mathtt{N}\}\right]\!\right] : R^{\underline{\mathtt{y}}} \to R. \qquad ([\![\beta]\!])$$

Proof. See Section 3. □

The one above is a fundamental property, because, as we are going to see in the next section, it gives an operational meaning to λ-term-with-context-variabless as actual programs. It is so important that it makes sense to only consider the "spaces" X (in any kind of sense) that are equipped with a map $[\![\_]\!] : \Lambda^{\vdash} \to X$ that satisfy the equation ($[\![\beta]\!]$). We call the data $(X, [\![\_]\!])$ a *(denotational) model of the λ-calculus.* In fact, whenever we dispose, in the very general sense of category theory, of a retraction (λ, fun) on a space X (like we have for $X := R$), then one can canonically define $[\![\_]\!]$ following the same definition that we gave, and this will automatically satisfy ($[\![\beta]\!]$). The interpretation map in R defined above gives one particular model, called the *Plotkin-Scott-Engeler graph model*[6].

We defined λ-terms precisely with the hope that they would be related, in some sense, the RE sets. Indeed, in analogy with Theorem 3.11 of the notes of Lecture 1, λ-terms only define a very special kind of functions:

Proposition 2.6. *The interpretation of any closed λ-term-with-context-variables is RE.*

Definition 2.7. *We say that a continuous $f : R \to R$ is* λ-computable *iff $\lambda(f) \subseteq \mathbb{N}$ is RE.*

Ex. 1 — Remember the functions Y and *fun* from Lecture 1. Find a closed term $\mathtt{Y}$ such that $[\![\vdash \mathtt{Y}]\!] = \lambda(Y \circ \mathit{fun})$. Conclude that the function $Y \circ \mathit{fun} : R \to R$ is λ-computable.

Answer (Ex. 1) — See section 4. □

Proposition 2.8. *The interpretation $[\![\underline{\mathtt{x}} \vdash \mathtt{M}]\!] : R \to R$ of a λ-term-with-context-variables is λ-computable.*

[4]In the λ-abstraction case below, we can always invoke (α) in order to chose a $\mathtt{z}$ with those properties. Indeed, $\mathtt{M}, \mathtt{N}, \mathtt{x}$ are fixed and $\mathtt{z}$ pops out by decomposing $\mathtt{M}$, so we are free to choose it as it pleases us.

[5]The reason why one prefers the inductive definition above to the geometric one is because, due to the inductive nature of λ-terms, in practice the great majority of the proofs are done by induction and then one needs the inductive characterisation above.

[6]There are several similar but different variants of this construction.

At this point, one could develop all the relevant notions of computability theory in this setting. Most notably, one can exhibit a closed term U that plays the same role of a universal Turing machine, in the sense that $\{@(\llbracket U \rrbracket, \{n\}) \mid n \in \mathbb{N}\}$ is an enumeration of all the RE sets.

Remark 2.9. *Note how the denotation semantics in R that we sketched here provides thus a first sense in which we can understand the λ-calculus as a programming language: 1) the functions (in R) that it defines are not any function, but only λ-computable ones (as Turing Machines only define computable functions on $\mathbb{N}$), and 2) it is universal for RE sets (as Turing machines are universal for computable functions on $\mathbb{N}$).*

Remark 2.10 (Denotational Semantics)**.** *From a PL perspective, the definition of an interpretation map is said to give a* denotational semantics *(DS, for short) for programs (in this case, the λ-terms-with-context-variables). This can be seen as a way of specifying which entities programs (which are just pieces of syntax) can be thought of manipulating, by providing a way of interpreting programs as mathematical entities that* compose *and that can be* built inductively *as them. Thus, answering to the question "*which *mathematical entity does a given program implement?"*[7] *One constraint typically required is for those entities to abstract as much as possible from any implementation detail*[8]*. More about DNS in Remark 2.20.*

Remark 2.11. *The notion of Scott-continuity and its properties that we have mentioned for R, are actually crucial aspects of the majority of the DNS's of the λ-calculus.*

2.2 Introduction to Operational semantics (full β-reduction)

We first encountered (β) ; we then transformed it into the crucial defining property of denotational semantics $(X, \llbracket\_\rrbracket : \Lambda^{\vdash} \to X)$, holding for the interpretation of terms-with-context-variables: equation ($\llbracket\beta\rrbracket$). Now, the interpretation map of any denotational model induces an equivalence on $\Lambda^{\vdash}$ that equates, at least, all the terms-with-context-variables $\underline{\mathtt{y}} \vdash (\lambda\mathtt{x}.\mathtt{M})\,\mathtt{N}$ with $\underline{\mathtt{y}} \vdash \mathtt{M}\{\mathtt{x} := \mathtt{N}\}$, if $\underline{\mathtt{y}} \vdash \mathtt{N}$. We are going now to make this into an equivalence (and even more: a dynamics) on actual terms (without context variables). This even simplifies the handling of variables and shows that, actually, this equivalence is very natural. But, most importantly, it gives another way, even simpler, of understanding λ-calculus as a programming language: this time the programs will be the λ-terms (without context variables), and their behaviour as programs is given in a more syntactic way.

So, let us forget about context variables, and only focus on λ-terms. Then ($\llbracket\beta\rrbracket$) suggests to consider an equivalence on λ-terms, called β-equivalence:

$$(\lambda\mathtt{x}.\,\mathtt{M})\,\mathtt{N} =_\beta \mathtt{M}\{\mathtt{x} := \mathtt{N}\}. \tag{$=_\beta$}$$

Actually, it is more natural to orient the equation, called β-reduction:

$$(\lambda\mathtt{x}.\,\mathtt{M})\,\mathtt{N} \to_\beta \mathtt{M}\{\mathtt{x} := \mathtt{N}\} \tag{$\to_\beta$}$$

This orientation has a clear meaning in terms of a dynamics of *function computation as formal substitution*. We can then retrieve ($=_\beta$) as a quotient with respect to the reflexive, symmetric and transitive closure of ($\to_\beta$).

[7] This also justifies the expression "semantics": it clarifies how can we assemble those pieces of syntax in order to program within the language some entity that we have in mind; therefore, sometimes people take a Fregean-like philosophical perspective and summarise this by saying that the DNS is a way of providing a "meaning" to programs.

[8] In the same way as in, say, linear algebra, we are interested in the abstract manipulations of vectors, not in their implementation as "actual arrows".

Remark 2.12. *We could have even started the whole topic by* $(\to_\beta)$ *with the idea of* function computation as formal substitution*, then obtain* $(=_\beta)$*, and finally introduce* $(\llbracket\beta\rrbracket)$*. Here we did the opposite: from* $(\llbracket\beta\rrbracket)$*, to* $(=_\beta)$ *to* $(\to_\beta)$*. This is only a matter of taste.*

Notice that for the quotient wrt $(=_\beta)$ to make sense, we need to say how are we going to extend this relation to terms which are not of the particular shape written in $(\to_\beta)$. Algebraically speaking, we need to extend the equivalence to a *congruence*[9]. Sometimes, one wants to restrict $(=_\beta)$ (or $(\to_\beta)$) to terms under certain constraints (and then still extend it to a congruence). In all cases, the obtained congruences are called *λ-theories*.

Remark 2.13 (Operational Semantics)**.** *From a PL perspective, the definition of a dynamics on programs is said to give an* operational semantics *(OPS, for short) for them (in this case, λ-terms). This can be seen as a way of specifying how the programs (which are just pieces of syntax) concretely behave as computational objects, this is typically done in a machine-inspired way. In some cases (like the λ-calculus, or more generally purely functional languages), the OPS can be described as a reduction relation on programs (like* $(\to_\beta)$*). The induced congruence from the dynamics (like* $(=_\beta)$*) is said to give a* operational equivalence[10]*. A λ-theory is, thus, the choice of an operational equivalence on λ-terms. More about OPS in Remark 2.20.*

Now, there are several different notions of congruence, which in turn go with several restrictions on $(=_\beta)$, giving rise to many different λ-theories[11]. Some of them make particular sense from a programming language perspective. So, similarly to how there are different DNS's, there are different OPS too, i.e. different possible choices of defining what it means for a term to run.

For λ-calculus, a natural one (and vaguely close to real programming languages) is obtained by understanding Λ as being the instruction set for an abstract machine, which implements what is called a *weak head-reduction* execution mode. This approach is also taken in classical realisability (see Lecture 5). The congruence that one uses for defining this OPS can be also defined by what are called *head-contexts*. There are other OPS's that are closer to real implementations, such as the so-called "call-by-value" or the more refined "call-by-need", etc. Those often seem to have similar definitions, but turn out to define very different λ-theories. Here however, we will stay much simpler and take the easiest OPS: this is definitely far from real implementations, but still it is crucial for theoretical studies about the semantics of λ-calculus, and it is the prototypical one:

Definition 2.14. *The* full β-reduction $\to_\beta \subseteq \Lambda \times \Lambda$ *is defined by the following inductive rules:*

$$\frac{}{(\lambda \mathtt{x}.\,\mathtt{M})\,\mathtt{N} \to_\beta \mathtt{M}\{\mathtt{x} := \mathtt{N}\}} \qquad \frac{\mathtt{M} \to_\beta \mathtt{N}}{\lambda \mathtt{x}.\mathtt{M} \to_\beta \lambda \mathtt{x}.\mathtt{N}} \qquad \frac{\mathtt{M} \to_\beta \mathtt{M}'}{\mathtt{M}\,\mathtt{N} \to_\beta \mathtt{M}'\,\mathtt{N}} \qquad \frac{\mathtt{N} \to_\beta \mathtt{N}'}{\mathtt{M}\,\mathtt{N} \to_\beta \mathtt{M}\,\mathtt{N}'}$$

Let $\twoheadrightarrow_\beta$ *be the reflexive and transitive closure of* $\to_\beta$*. Let* $=_\beta$ *be the symmetric closure of* $\twoheadrightarrow_\beta$ *(i.e. its induced equivalence, or equivalently of* $\to_\beta$*). A term* $\mathtt{M}$ *is* normal *if there is no possible* $\to_\beta$*-reduction from* $\mathtt{M}$*. We say that a normal* $\mathtt{N}$ *is a* normal form of $\mathtt{M}$ *if* $\mathtt{M} \twoheadrightarrow_\beta \mathtt{N}$*, and in that case we say that* $\mathtt{M}$ *has a terminating reduction (on* $\mathtt{N}$*). Terms of shape* $(\lambda\mathtt{x}.\,\mathtt{M})\,\mathtt{N}$ *are called* redexes.

Ex. 2 — Find a normal form form, if any, of the following terms: $\mathtt{I}\,\mathtt{x}$, $(\lambda \mathtt{xy.x})\,\mathtt{y}$, $(\lambda \mathtt{xy.y})\,\mathtt{y}$, $(\lambda \mathtt{xy.z})\,\mathtt{M}\,\mathtt{N}$, $(\lambda \mathtt{xyz.(xz)(yz)})\,\mathtt{z}\,\mathtt{x}\,\mathtt{y}$.

Answer (Ex. 2) — Simply follow the definition. □

[9] I.e., an equivalence compatible with some algebraic structure.
[10] In practice, operational equivalence and operational semantics are also used as synonyms.
[11] A continuum of them!

The abstract rewriting system $(\Lambda, \to_\beta)$ is clearly not deterministic, because there is a term with at least two different reductions, for instance: $\mathtt{x}(\mathtt{Iy})\ {}_\beta\!\leftarrow (\mathtt{Ix})(\mathtt{Iy}) \to_\beta \mathtt{Ixy}$. Remark that a necessary condition for a term in order to reduce to two different terms is to have two redexes, but the condition is not sufficient: for example, $\mathtt{I(II)}$ has two different redexes, but it only reduces to one term: $\mathtt{I(II)} \to_\beta \mathtt{II}$. This is because we defined $\to_\beta$ as a relation (i.e. a pair). Looking at the different reduction paths would correspond to taking into account the different derivations of Definition 2.14, but we will not do this.

Remark 2.15. *There is a term with no normal form. For instance, the Ω of Exercise 4 (cfr. Figure 1). There is also a term which has a normal form but not all its reductions terminate. For instance, $(\lambda\mathtt{x}.\mathtt{I})\,\Omega$. Finally, notice that being $=_\beta$ does not imply that one reduces to the other. For instance, $(\lambda\mathtt{x}.\mathtt{I})\,\Omega =_\beta \mathtt{II}$, because both $\to_\beta$-reduce to $\mathtt{I}$, but $(\lambda\mathtt{x}.\mathtt{I})\,\Omega \not\to_\beta \mathtt{II}$ and $\mathtt{II} \not\to_\beta (\lambda\mathtt{x}.\mathtt{I})\,\Omega$.*

In fact, a fundamental result due to Church and Rosser says that to be $=_\beta$ is exactly the same as $\twoheadrightarrow_\beta$-reducing to a common term. One can immediately show that this fact is actually equivalent to another natural notion, which is a weak but powerful form of determinism, called *confluence*. It is in this form that one usually states it:

Theorem 2.16 (Church, Rosser)**.** *The abstract rewriting system $(\Lambda, \to_\beta)$ is confluent.*

Ex. 3 — Prove that, as immediate corollary to the confluence above, any λ-term has at most one normal form (that is, in any β-equivalence class there is at most one normal term).

Answer (Ex. 3) — See Section 4. □

Ex. 4 — Let $\delta := \lambda\mathtt{x}.\mathtt{x}\,\mathtt{x}$ and $\Omega := \delta\,\delta$ (cfr. Figure 1). Show that: $\Omega \to \Omega$
Let $\Delta_\mathtt{M} := \lambda\mathtt{x}.\,\mathtt{M}(\mathtt{x}\,\mathtt{x})$. For $\mathtt{f}$ a variable, does $\Delta_\mathtt{f}\,\Delta_\mathtt{f}$ have a normal form?
Show that $\Delta_\mathtt{I}\,\Delta_\mathtt{I} \twoheadrightarrow \Delta_\mathtt{I}\,\Delta_\mathtt{I}$.
Let $\mathtt{Y} := \lambda\mathtt{f}.\,\Delta_\mathtt{f}\Delta_\mathtt{f}$ (cfr. Figure 1). Show that $\mathtt{Y}\,\mathtt{M} =_\beta \mathtt{M}\,(\mathtt{Y}\,\mathtt{M})$.
Let $\Phi := \lambda\mathtt{xy}.\,\mathtt{y}(\mathtt{x}\,\mathtt{x}\,\mathtt{y})$ and $\Theta := \Phi\,\Phi$. Show that $\Theta\,\mathtt{M} \twoheadrightarrow \mathtt{M}\,(\Theta\,\mathtt{M})$.
Let $\Psi := \lambda\mathtt{xy}.\,\mathtt{y}(\mathtt{x}\,\mathtt{x})$ and $\chi := \Psi\,\Psi$. Show that $\chi\,\mathtt{M} \twoheadrightarrow \mathtt{M}\,\chi$ and $\chi \twoheadrightarrow \lambda\mathtt{x}.\,\mathtt{x}\,\chi$.
Are you able to keep reducing χ? Does χ have a normal form?
Let $W_\mathtt{M} := \lambda\mathtt{xy}.\mathtt{M}(\mathtt{x}\,\mathtt{x}\,\mathtt{y})$ and, for $\mathtt{f}$ a variable, $\Xi_\mathtt{z} := \lambda\mathtt{f}.W_\mathtt{f}W_\mathtt{f}\mathtt{z}$. Show that $\Xi_\mathtt{z}\,\mathtt{M} =_\beta \mathtt{M}\,(\Xi_\mathtt{z}\,\mathtt{M})$.
Let $\mathtt{A} := \Theta(\lambda\mathtt{xy}.(\lambda\mathtt{z}.\mathtt{z}(\mathtt{x}\,\mathtt{y})))$. Show that, for $\mathtt{x} \neq \mathtt{z}$, $\mathtt{A}\,\mathtt{z} \twoheadrightarrow \lambda\mathtt{x}.\,\mathtt{x}(\mathtt{A}\,\mathtt{z})$.
Are you able to keep reducing $\mathtt{Az}$? Does $\mathtt{Az}$ have a normal form?

Answer (Ex. 4) — Carefully follow the definition of the reduction. □

Definition 2.17. *A term $\mathtt{F}$ such that $\mathtt{F}\,\mathtt{M} =_\beta \mathtt{M}\,(\mathtt{F}\,\mathtt{M})$ for all term $\mathtt{M}$, is called a* fixed point combinator.

Note that Exercise 4 shows that $\mathtt{Y}$ and Θ are fixed point combinators[12], as well as $\Xi_\mathtt{N}$, as soon as $\mathtt{N}$ is closed.

Remark 2.18. *If you were able to successfully do Exercise 2 and Exercise 4, then you understood substitution, reduction, and α-conversion: congratulations!*

Remark 2.19. *The notion of Scott-continuity that we have seen in R is actually reflected in the OPS's of the λ-calculus itself: for instance, λ-terms are Scott-continuous in a very precise sense. Many more aspects of domain theory crucially hold for the λ-calculus at the level of its OPS's.*

[12] $\mathtt{Y}$ is called Curry's combinator, and Θ Turing's.

Remark 2.20 (On Denotational and Operational Semantics). *We give below some ways in which DS and OPS (and their relationship) can be understood:*

Conceptually: *As the study of the mathematical entities that* look like *like programs (of the particular language under study)*[13]*. Sometimes, those entities are functions, as one would expect. For example, it is the case for our model R. However, interpreting terms as functions is typically subtle: For example, in λ-calculus, we know that in order to be able to define an interpretation in a set X, its function space X^X needs to retract onto it (i.e. we need (β) to hold). This cuts out many mainstream mathematical "functional" structures, and one usually deals with function spaces based on domain theory (like we did for R). But terms can, for example, be equally well interpreted as relations, and these kind of interpretations are also very natural (especially when considering Linear Logic and its differential aspects). In fact, the general framework is given by reflexive objects living in (weak) Cartesian closed categories ((w)CCC, for short), with terms being endomorphisms on it. Living in a CCC actually gives rise to a stronger notion of denotational semantics, in that it not only validates ($[\![\beta]\!]$) but also $[\![\eta]\!]$, making the semantics* extensional[14]*. Only requiring weak CCC's gives instead a non-extensional semantics (only ($[\![\beta]\!]$) holds), but the research community devoted most of the study to the case of CCC's. We will give a quick overview of it in the next lesson.*

Mathematically: *One usually fixes an equivalence between programs (typically induced by a OPS sound wrt the DNS). Then DS is a way of defining* invariants with respect to the equivalence under study*. This is analogous to, say, homotopy/homology groups in algebraic topology: not because it has to do with groups or geometry, but because it associates each program with an abstract entity which is compatible with (read: invariant with respect to) the equivalence under study*[15]*. Moreover, the OPS usually gives rise to an oriented relation $\to_\beta$ (giving rise to a dynamical system). Therefore, in particular, DNS defines* invariants of the dynamics*, in the same way that physics defines conservation lawsof dynamical systems. Observe that different DNS's give different invariants; in physics, Noëther's Theorem says that a certain class of symmetries define invariants. In λ-calculus, it is the class of reflexive objects in (w)CCC's that defines invariants*[16]*.*

In practice: *DNS and OPS are often used as a way of telling two programs not equivalent wrt the respective equivalences, by showing that the associated invariants (their interpretation) is different. If the DNS and the OPS interact well (i.e. the equivalences they induce are related in some way), then the DNS may be useful as a way of abstracting away details about programs that are not relevant for the situation under study, in order to move to a framework in which more suited tools are available.*

Mathematics for Informatics *If an "operational semantics" (OPS, for short) is given (see Section 2.2), one also typically wonders if the OPS and the DS interact well with each other*[17]*. The idea here is to study whether some operational properties (for instance, the fact of a program performing sequential operations on a machine), can be described in a purely mathematical way, abstracting away from a machine*[18]*.And, vice-versa, if some abstract property has a operational (or machine dependent) counterpart*[19]*.*

[13] In analogy to how standard linear algebra studies mathematical entities which look like our physical space.
[14] As it equates programs whenever they behave the same as functions.
[15] For us, $=_\beta$ (Definition 2.14), or refinements of it; in algebraic topology, homeomorphism or refinements of it.
[16] At least for the extensional case, one can also reasonably say that this is the class of *all* the invariants.
[17] Asking at least for a property called "adequacy", and hoping for another called "full abstraction".
[18] See work on continuity –> stability –> sequentiality.
[19] See parallel-or etc

2.3 Some basic and fun(ctional) programming

Remember that, for us, β-equivalent programs are operationally equivalent. A normal form plays the role of a result (a completely defined output).

Remark that we do not have any built-in data structure, nor type system of any kind, so the user has to implement them all from scratch.

We will only implement Booleans and natural numbers data types, but we give no type system for now. However, one could do data structures, computable analysis, and in general implement all computable functions, since, as we will mention, Λ is Turing-complete.

Remark that we already have a notion of "function implemented by a term", and it is the one given by denotational semantics in, say, R. Here we take another, standard, notion of implementation of a function which is not based on the structure of the program (as in denotational semantics), but on its operational behaviour: we say that a function is implemented by a term if the term behaves, in the sense of the OPS, like the function on all inputs.

Definition 2.21. *If we think of a datatype as a set A together with some (total) operations $f : A^n \to A$, then in order to implement it we need to choose an (injective) encoding $\ulcorner . \urcorner : A \to \Lambda$ of its elements and, for each f, choose a closed $\ulcorner f \urcorner \in \Lambda$ satisfying the following specification: for all $\vec{a} \in A^n$, we have*[20] $\ulcorner f \urcorner \ulcorner \vec{a} \urcorner =_\beta \ulcorner f(\vec{a}) \urcorner$. *We say in this case that f is* implementable in the λ-calculus (wrt the chosen implementation of the datatype).

Remark that, since we really working in the quotient in the definition above, the specification above equivalently gives $\ulcorner f \urcorner \vec{\mathtt{M}} =_\beta \ulcorner f(\vec{a}) \urcorner$ whenever $\mathtt{M}_i =_\beta \ulcorner a_i \urcorner$ (for all i).

Ex. 5 — The usual encoding of Booleans, due to Church, uses projection terms in order to encode the two boolean values:

$$\text{TRUE} := \lambda \mathtt{xy}.\,\mathtt{x} \qquad \text{FALSE} := \lambda \mathtt{xy}.\,\mathtt{y}$$

Implement the boolean functions *not, and, or, xor.*

Answer (Ex. 5) — See Section 4. □

* **Ex. 6** — The usual encoding of natural numbers, due to Church, is in unary base, using iterators:

$$\ulcorner n \urcorner := \lambda \mathtt{fx}.\,\mathtt{f}(\overset{(n)}{\cdots}(\mathtt{f}\,\mathtt{x}))$$

Thus, for example, $\ulcorner 0 \urcorner = \text{FALSE}$, $\ulcorner 1 \urcorner = \lambda \mathtt{fx}.\,\mathtt{f}\,\mathtt{x}$, $\ulcorner 5 \urcorner = \lambda \mathtt{fx}.\,\mathtt{f}(\mathtt{f}(\mathtt{f}(\mathtt{f}(\mathtt{f}\,\mathtt{x}))))$. Implement the *successor* function, *addition*, *test to zero* (return true if the argument is zero, false otherwise).

Answer (Ex. 6) — See section 4. □

One can also implement the (truncated[21]) predecessor function, with the following specification: $\text{PRED}\ulcorner n \urcorner =_\beta \begin{cases} \ulcorner n-1 \urcorner & \textit{if } n > 0 \\ \ulcorner 0 \urcorner & \textit{if } n = 0. \end{cases}$ But this is not straightforward (with Church encoding): one way to do it is to use the data-structure of pairs: starting from the pair $(0, 1)$, recursively add 1 to both components until the second one is n, then return the first one. For lack of space, we will not show how to do it here, but we will use it to implement other interesting functions.

[20] Remark that we are using here the β-equality, but one may want to be more subtle and use other λ-theories.

[21] Be careful, because while the truncated predecessor satisfies, as usual, $\text{PRED}\,\text{SUCC}\ulcorner n \urcorner =_\beta \ulcorner n \urcorner$ for all $n \in \mathbb{N}$, it only satisfies $\text{SUCC}\,\text{PRED}\ulcorner n \urcorner =_\beta \ulcorner n \urcorner$ if $n > 0$. For $n = 0$ we have $\text{SUCC}\,\text{PRED}\ulcorner 0 \urcorner =_\beta \ulcorner 1 \urcorner$.

* **Ex. 7** — Supposing to have a term PRED $\in \Lambda$ that implements the predecessor function as above, implement the (truncated) *subtraction* of the first argument minus the second, with the following specification: $\textsc{subtr}\ulcorner n \urcorner \ulcorner m \urcorner =_\beta \begin{cases} \ulcorner n-m \urcorner & \textit{if } n-m>0 \\ \ulcorner 0 \urcorner & \textit{if } n-m \leq 0. \end{cases}$ Moreover, implement the *less than or equal* test (returns true if the first argument is smaller than or equal to the second argument, and false otherwise), implement the *equality test* (returns true if the two arguments are two equal numbers, false otherwise) and implement the *strictly less than* test (returns true if the first argument is strictly smaller than the second argument, and false otherwise).

Answer (Ex. 7) — See section 4. □

Let us now see how to implement primitive recursion. This has to do with fix-points.

Theorem 2.22. *Let $h : \mathbb{N} \times \mathbb{N} \times \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$. Let $\mathrm{aux}_h^g : \mathbb{N}^{\mathbb{N}^{n+1}} \to \mathbb{N}^{\mathbb{N}^{n+1}}$ defined by $\mathrm{aux}_h^g(f)(0,\vec{m}) := g(\vec{m})$ and $\mathrm{aux}_h^g(f)(n+1,\vec{m}) := h(f(n,\vec{m}),n,\vec{m})$.*

Then aux_h^g admits a fixed-point, namely the function $F : \mathbb{N}^{n+1} \to \mathbb{N}$ inductively defined by $F(0,\vec{m}) := g(\vec{m})$ and $F(n+1,\vec{m}) := h(F(n,\vec{m}),n,\vec{m})$.

Moreover, if g, h are implemented by closed terms $\ulcorner g \urcorner, \ulcorner h \urcorner \in \Lambda$, then F is implemented by the closed term $\mathtt{Y}\,\mathtt{A} \in \Lambda$, where $\mathtt{A} := \lambda \mathtt{f}\,\mathtt{n}\,\vec{\mathtt{m}}.\, \textit{ISZERO}\,\mathtt{n}\,(\ulcorner g \urcorner \vec{\mathtt{m}})\,(\ulcorner h \urcorner\,(\mathtt{f}\,(\textit{PRED}\,\mathtt{n})\,\vec{\mathtt{m}})\,(\textit{PRED}\,\mathtt{n})\,\vec{\mathtt{m}})$.

Proof. That F is a fixed-point of aux_h^g is immediate. Let us now see why $\mathtt{Y}\,\mathtt{A}$ implements it. For this, we have to show that $\mathtt{Y}\,\mathtt{A}\ulcorner n \urcorner \ulcorner \vec{m} \urcorner =_\beta \ulcorner F(n,\vec{m}) \urcorner$ for all $(n,\vec{m}) \in \mathbb{N}^{n+1}$. We show it by induction on $n \in \mathbb{N}$. We have:

$$\begin{aligned} \mathtt{Y}\,\mathtt{A}\ulcorner n \urcorner \ulcorner \vec{m} \urcorner &=_\beta \mathtt{A}\,(\mathtt{Y}\,\mathtt{A})\ulcorner n \urcorner \ulcorner \vec{m} \urcorner \\ &=_\beta \mathrm{ISZERO}\ulcorner n \urcorner (\ulcorner g \urcorner \ulcorner \vec{m} \urcorner)\,(\ulcorner h \urcorner ((\mathtt{Y}\,\mathtt{A})\,(\mathrm{PRED}\ulcorner n \urcorner)\ulcorner \vec{m} \urcorner)\,(\mathrm{PRED}\ulcorner n \urcorner)\ulcorner \vec{m} \urcorner) \\ &=_\beta \mathrm{ISZERO}\ \ulcorner n \urcorner\ \ulcorner g(\vec{m}) \urcorner\ (\ulcorner h \urcorner ((\mathtt{Y}\,\mathtt{A})\ulcorner n-1 \urcorner \ulcorner \vec{m} \urcorner)\ulcorner n-1 \urcorner \ulcorner \vec{m} \urcorner) \end{aligned}$$

Now if $n = 0$, we continue the equalities as:

$$=_\beta \ulcorner g(\vec{m}) \urcorner =_\beta \ulcorner F(0,\vec{m}) \urcorner.$$

If $n \geq 1$, we continue the equalities as:

$$\begin{aligned} &=_\beta \ulcorner h \urcorner ((\mathtt{Y}\,\mathtt{A})\ulcorner n-1 \urcorner \ulcorner \vec{m} \urcorner)\ulcorner n-1 \urcorner \ulcorner \vec{m} \urcorner \\ &=_\beta \ulcorner h \urcorner \ulcorner F(n-1,\vec{m}) \urcorner \ulcorner n-1 \urcorner \ulcorner \vec{m} \urcorner \\ &=_\beta \ulcorner h(F(n-1,\vec{m}),n-1,\vec{m}) \urcorner \\ &=_\beta \ulcorner F(n,\vec{m}) \urcorner. \end{aligned}$$

□

The Theorem above says that implementing a primitive recursion is not harder than implementing its building blocks. For instance, let us implement the factorial: $(\_)! : \mathbb{N} \to \mathbb{N}$, $0! := 1$ and $(n+1)! := n!\,(n+1)$.

This can be written as a primitive recursive function via $g := 1 \in \mathbb{N}$ and $h : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, $h(a,b) := a(b+1)$. Those are clearly implementable by $\ulcorner 1 \urcorner$ and $\ulcorner h \urcorner := \lambda \mathtt{a}\,\mathtt{b}.\,\mathrm{MULT}\,\mathtt{a}\,(\mathrm{SUCC}\,\mathtt{b})$, respectively. Therefore, the theorem ensures that $\mathtt{Y}\,(\lambda \mathtt{f}\,\mathtt{n}.\,\mathrm{ISZERO}\,\mathtt{n}\ulcorner 1 \urcorner (\ulcorner h \urcorner (\mathtt{f}\,(\mathrm{PRED}\,\mathtt{n}))\,(\mathrm{PRED}\,\mathtt{n})))$ implements $(\_)!$. Developing $\ulcorner h \urcorner$ we can also obtain the slightly more readable:

$$\mathtt{Y}\,(\lambda \mathtt{f}\,\mathtt{n}.\,\mathrm{ISZERO}\,\mathtt{n}\ulcorner 1 \urcorner (\mathrm{MULT}\,(\mathtt{f}\,(\mathrm{PRED}\,\mathtt{n}))\,\mathtt{n}))$$

where we have also used the fact that $\mathrm{SUCC}\,(\mathrm{PRED}\ulcorner n \urcorner) =_\beta \ulcorner n \urcorner$ if $n \geq 1$.

One can do much more than just the previous encodings and implementations. For instance, one can encode data-structures such as pairs, lists, trees, but also rational numbers, computable real numbers, etc.

For arithmetic[22], we have basically seen that we can implement all primitive recursive functions. Actually, one can also implement minimisation (but we are not going to prove it), and therefore get all partial recursive functions:

Theorem 2.23. *Taking, say, Church encoding of natural numbers, then the* partial *functions* $\mathbb{N} \to \mathbb{N}$ *which are implementable*[23] *in λ-calculus are exactly the Turing-computable ones (i.e. the partial recursive functions).*

Thus we say that the (untyped) λ-calculus is a *Turing-complete programming language* and supports Church-Turing's thesis, which says that *we found the satisfying mathematical formalisation of the intuitive notion of computable partial function $\mathbb{N} \to \mathbb{N}$, namely one which is programmable in the λ-calculus wrt some*[24] *implementation of the datatype $\mathbb{N}$ (or, equivalently, on a Turing machine wrt to some implementation of the datatype $\mathbb{N}$).* Of course we can actually use any encoding we like, as long as we have translations with Church's one: this is just like in Turing Machines (or any other model of computation), where we first need to fix an encoding of natural numbers on the alphabet of the machine. Remark that, while Turing-machines offer an imperative programming paradigm, λ-calculus offers a functional paradigm.

Remark 2.24. *Beware that, as always, being Turing-complete does* not *mean that we can implement* all *algorithms for computing a computable function, but only that for any computable function we can implement* at least one *algorithm computing it! More specifically, deep results in λ-calculus*[25] *show that the only kind of computation that Λ implements is necessarily "sequential"*[26] *(this can, of course, be made rigorous). For example, one cannot program in Λ the parallel-or algorithm: this is the one which computes the boolean or of two inputs by running them in parallel and inspecting first the first which stops (if any). This corresponds to a partial or function, which of course we can implement (we already did:* OR*), since Λ is Turing-complete; but we can implement it in a sequential way, not in the parallel way described above!*

* **Ex. 8** — Remember the Euclidean division: there are functions $q, r : \mathbb{N} \times \mathbb{N}_{>0} \to \mathbb{N}$ such that, for all $m \in \mathbb{N}$, $n \in \mathbb{N}_{>0}$, we have $m = n \cdot q(m,n) + r(m,n)$ and $r(m,n) < n$.

1. Prove that the following inductive characterisation of q, r holds:

$$\begin{cases} r(0,n) = 0 & \\ r(m+1,n) = r(m,n)+1 & \textit{if } r(m,n)+1 < n \\ r(m+1,n) = 0 & \textit{if } r(m,n)+1 = n \end{cases} \qquad \begin{cases} q(0,n) = 0 & \\ q(m+1,n) = q(m,n) & \textit{if } r(m,n)+1 < n \\ r(m+1,n) = q(m,n)+1 & \textit{if } r(m,n)+1 = n \end{cases}$$

2. Implement q and r in λ-calculus.

[22]Remark that this is enough because, we can encode "anything" as natural numbers. This is the exact same reason why when working with Turing-machines one can restrict its attention to numerical functions encoded via, say, a binary alphabet.

[23]One should precisely define what does it mean to implement a *partial* function on a datatype (remark that our previous definition only considers total operations). The problem is that of understanding how to characterise the terms whose run produces *no observable output* whatsoever (these terms are sometimes also called "meaningless"). This is typically done via the notion of *unsolvable* term, which is related with another λ-theory "$=_{\mathrm{BT}}$", and one works then in the quotient $\Lambda/_{\mathrm{BT}}$.

[24]Here we have seen the most natural one, Church's.

[25]Those results are non-trivial and require the developing some theory of "approximation" of λ-terms: either on the style of the one mentioned at the beginning of Lecture 1 based on Scott continuity, or by a completely different approach based on the notion of differentiation and coming from Linear Logic.

[26]In a very loose sense, think of, say, one-tape deterministic Turing-machines.

Answer (Ex. 8) — See Section 4. [*Hint: for 1), you need to prove that i) if q, r are the Euclidean division then they satisfy the equations above, and ii) if q, r are defined by the equations above, then they are total and compute the Euclidean division.*] $\square$

3 Appendix

Proofs of Lemma 1.3, Proposition 1.4 and Lemma 1.6

First, we need a couple of lemmas about simultaneous substitution. Remember the definition:

Definition 3.1. *Given a bijection σ with domain $\underline{\mathtt{x}}$, we define the simultaneous σ-renaming $\mathtt{M}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}$ of a λ-term $\mathtt{M}$ by indution as:*

$$\mathtt{x}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\} = \sigma(\mathtt{x}) \qquad \mathtt{y}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\} = \mathtt{y} \textit{ if } \mathtt{y} \notin \underline{\mathtt{x}}$$

$$(\mathtt{P}\,\mathtt{Q})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\} := (\mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\})(\mathtt{Q}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\})$$

$$(\lambda\mathtt{z}.\mathtt{P})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\} = \lambda\mathtt{z}.(\mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}) \textit{ if } \mathtt{z} \notin \underline{\mathtt{x}} \cup \sigma(\underline{\mathtt{x}}).$$

Lemma 3.2. *Let $\sigma : \underline{\mathtt{x}} \to \underline{\mathtt{z}}$ and $\mu : \underline{\mathtt{y}} \to \underline{\mathtt{w}}$ be bijections. If $\underline{\mathtt{x}} \cap \underline{\mathtt{y}} = \emptyset$ and $\underline{\mathtt{x}} \cap \underline{\mathtt{w}} = \emptyset$, then:*

$$\mathtt{M}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}}), \underline{\mathtt{y}} := \mu(\underline{\mathtt{y}})\} = \mathtt{M}\{\underline{\mathtt{y}} := \mu(\underline{\mathtt{y}})\}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}.$$

Proof. Induction on $\mathtt{M}$.

Case $\mathtt{y}$: then the LHS is $\mu(\mathtt{y})$ and the RHS is $\mu(\mathtt{y})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\} = \mathtt{y}$ because $\underline{\mathtt{y}} \cap \underline{\mathtt{x}} = \emptyset$.

Case $\mathtt{x}$: then the LHS is $\sigma(\mathtt{x})$ and the RHS is $\mathtt{x}\{\underline{\mathtt{y}} := \mu(\underline{\mathtt{y}})\}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\} = \mathtt{x}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\} = \sigma(\mathtt{x})$ because $\underline{\mathtt{y}} \cap \underline{\mathtt{x}} = \emptyset$.

Case $\mathtt{P}\,\mathtt{Q}$: immediate by IH (used twice).

Case $\lambda\mathtt{v}.\mathtt{P}$: Wlog by ($\alpha$) we have $\mathtt{v} \notin \underline{\mathtt{x}} \cup \underline{\mathtt{y}} \cup \underline{\mathtt{z}} \cup \underline{\mathtt{w}}$. Therefore, the LHS is $\lambda\mathtt{v}.(\mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}}), \mathtt{y} := \mu(\underline{\mathtt{y}})\})$. The RHS is $(\lambda\mathtt{v}.\mathtt{P})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}\{\underline{\mathtt{y}} := \mu(\underline{\mathtt{y}})\} = (\lambda\mathtt{v}.(\mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}))\{\underline{\mathtt{y}} := \mu(\underline{\mathtt{y}})\} = \lambda\mathtt{v}.(\mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}\{\underline{\mathtt{y}} := \mu(\underline{\mathtt{y}})\})$, where in the first equality we used $\mathtt{v} \notin \underline{\mathtt{x}} \cup \underline{\mathtt{z}}$ and in the second one $\mathtt{v} \notin \underline{\mathtt{y}} \cup \underline{\mathtt{w}}$. Now the result follows from the IH. $\square$

Lemma 3.3. *For $\mathtt{y} \notin \underline{\mathtt{x}}$ and $\mathtt{y}' \notin FV(\mathtt{P}) \cup \underline{\mathtt{z}} \cup \underline{\mathtt{x}}$, we have*

$$\lambda\mathtt{y}'.(\mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}}), \mathtt{y} := \mathtt{y}'\}) = (\lambda\mathtt{y}.\mathtt{P})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}.$$

Proof.

$$\begin{array}{rcll} \lambda\mathtt{y}'.(\mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}}), \mathtt{y} := \mathtt{y}'\}) & = & \lambda\mathtt{y}'.(\mathtt{P}\{\mathtt{y} := \mathtt{y}'\}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}) & \textit{by Lemma 3.2 as } \mathtt{y}' \notin \underline{\mathtt{x}} \\ & = & (\lambda\mathtt{y}'.\mathtt{P}\{\mathtt{y} := \mathtt{y}'\})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\} & \textit{as } \mathtt{y}' \notin \underline{\mathtt{z}} \cup \underline{\mathtt{x}} \\ & = & (\lambda\mathtt{y}.\mathtt{P})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}. & \textit{as } \mathtt{y}' \notin FV(\mathtt{P}). \end{array}$$

$\square$

Now we can finally give the claimed proofs.

Proof of Lemma 1.3(σ-renaming). Induction on $\underline{\mathtt{x}} \vdash \mathtt{M}$.

Case $\dfrac{}{\underline{\mathtt{x}} \vdash \mathtt{x}}$: then $\dfrac{}{\underline{\mathtt{z}} \vdash \sigma(\mathtt{x})}$.

Case $\dfrac{\underline{\mathtt{x}} \vdash \mathtt{P} \quad \underline{\mathtt{x}} \vdash \mathtt{Q}}{\underline{\mathtt{x}} \vdash \mathtt{PQ}}$: then $\dfrac{\dfrac{\underline{\mathtt{x}} \vdash \mathtt{P}}{\underline{\mathtt{z}} \vdash \mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}}\sigma \quad \dfrac{\underline{\mathtt{x}} \vdash \mathtt{Q}}{\underline{\mathtt{z}} \vdash \mathtt{Q}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}}\sigma}{\underline{\mathtt{z}} \vdash (\mathtt{PQ})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}}$ where the σ-renamings are given by the IH.

Case $\dfrac{\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{P}}{\underline{\mathtt{x}} \vdash \lambda \mathtt{y}.\mathtt{P}}$, with $\mathtt{y} \notin \underline{\mathtt{x}}$: Let $\mathtt{y}' \notin FV(\mathtt{P}) \cup \underline{\mathtt{z}} \cup \underline{\mathtt{x}}$. Now we have:

$$\dfrac{\dfrac{\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{P}}{\underline{\mathtt{z}}, \mathtt{y}' \vdash \mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}}), \mathtt{y} := \mathtt{y}'\}}\sigma + [\mathtt{y} \mapsto \mathtt{y}']}{\underline{\mathtt{z}} \vdash (\lambda \mathtt{y}.\mathtt{P})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}}\lambda \mathtt{z}'$$

The $(\sigma + [\mathtt{y} \mapsto \mathtt{y}'])$-renaming is given by the IH and $(\sigma + [\mathtt{y} \mapsto \mathtt{y}'])$ is a bijection because $\mathtt{y} \notin \underline{\mathtt{x}}$ and $\mathtt{y}' \notin \underline{\mathtt{z}}$. The $\lambda \mathtt{y}'$-rule gives indeed $\lambda \mathtt{y}'.(\mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}}), \mathtt{y} := \mathtt{y}'\}) = (\lambda \mathtt{y}.\mathtt{P})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}$ thanks to Lemma 3.3. ☐

For the proof of the weakening, we need a remark which immediately follows from (σ)-renaming, but which is crucial:

Remark 3.4. *When we have* $\dfrac{\underline{\mathtt{x}}, \mathtt{z} \vdash \mathtt{P}}{\underline{\mathtt{x}} \vdash \lambda \mathtt{z}.\mathtt{P}}$ *(with* $\mathtt{z} \notin \underline{\mathtt{x}}$*), we can always Wlog assume that actually* $\mathtt{z} \notin \underline{\mathtt{x}} \cup \underline{\mathtt{y}}$*, where* $\underline{\mathtt{y}}$ *is any set of variables fixed* independently from $\mathtt{z}$ and $\mathtt{P}$.

Indeed, if it were not the case, we could take a $\mathtt{z}' \notin FV(\mathtt{P}) \cup \underline{\mathtt{x}} \cup \underline{\mathtt{y}}$ *and consider the bijection* $\mathrm{id}_{\underline{\mathtt{x}}} + [\mathtt{z} \mapsto \mathtt{z}'] : \underline{\mathtt{x}}, \mathtt{z} \to \underline{\mathtt{x}}, \mathtt{z}'$*. By renaming, we have* $\dfrac{\underline{\mathtt{x}}, \mathtt{z} \vdash \mathtt{P}}{\underline{\mathtt{x}}, \mathtt{z}' \vdash \mathtt{P}\{\mathtt{z} := \mathtt{z}'\}}(\mathtt{z} \mapsto \mathtt{z}')$*, and so we have* $\dfrac{\underline{\mathtt{x}}, \mathtt{z}' \vdash \mathtt{P}\{\mathtt{z} := \mathtt{z}'\}}{\underline{\mathtt{x}} \vdash \lambda \mathtt{z}.\mathtt{P}}\lambda \mathtt{z}'$ *with* $\mathtt{z}' \notin FV(\mathtt{P}) \cup \underline{\mathtt{x}} \cup \underline{\mathtt{y}}$*. We would then work with* $\mathtt{P}\{\mathtt{z} := \mathtt{z}'\}$ *instead of* $\mathtt{P}$ *and* $\mathtt{z}'$ *instead of* $\mathtt{z}$*. Notice that the result of applying the rule* $\lambda \mathtt{z}'$ *is indeed* $\lambda \mathtt{z}.\mathtt{P}$*, because* $\mathtt{z}' \notin FV(\mathtt{P})$*.*

Now we can continue with our proofs.

Proof of Lemma 1.3(Weakening). Induction on $\underline{\mathtt{x}} \vdash \mathtt{M}$.

Case $\dfrac{}{\underline{\mathtt{x}} \vdash \mathtt{x}}$: then $\dfrac{}{\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{x}}$.

Case $\dfrac{\underline{\mathtt{x}} \vdash \mathtt{P} \quad \underline{\mathtt{x}} \vdash \mathtt{Q}}{\underline{\mathtt{x}} \vdash \mathtt{PQ}}$: then $\dfrac{\dfrac{\underline{\mathtt{x}} \vdash \mathtt{P}}{\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{P}}W \quad \dfrac{\underline{\mathtt{x}} \vdash \mathtt{Q}}{\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{Q}}W}{\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{PQ}}$ with the weakenings given by IH.

Case $\dfrac{\underline{\mathtt{x}}, \mathtt{z} \vdash \mathtt{P}}{\underline{\mathtt{x}} \vdash \lambda \mathtt{z}.\mathtt{P}}$ with $\mathtt{z} \notin \underline{\mathtt{x}}$: Wlog by Remark 3.4, $\mathtt{z} \notin \{\mathtt{y}\} \cup \underline{\mathtt{x}}$. Now we have

$$\dfrac{\dfrac{\underline{\mathtt{x}}, \mathtt{z} \vdash \mathtt{P}}{\underline{\mathtt{x}}, \mathtt{z}, \mathtt{y} \vdash \mathtt{P}}W}{\underline{\mathtt{x}}, \mathtt{y} \vdash \lambda \mathtt{z}.\mathtt{P}}\lambda \mathtt{z}$$

The rule W is given by the IH. The rule $\lambda \mathtt{z}$ is applicable because $\mathtt{z} \notin \{\mathtt{y}\} \cup \underline{\mathtt{x}}$. ☐

Proof of Proposition 1.4. That a λ-term-with-context-variables is a λ-term is immediate, by induction on the derivation of any of its context variables. For the converse, we show by induction on $\mathtt{M} \in \Lambda$, that there is $\underline{\mathtt{x}}$ such that $\underline{\mathtt{x}} \vdash \mathtt{M}$.

Case $\mathtt{y}$: immediate.

Case $\mathtt{P}\,\mathtt{Q}$: By inductive hypotheses we have $\underline{\mathtt{x}} \vdash \mathtt{P}$ and $\underline{\mathtt{y}} \vdash \mathtt{Q}$. Then we have:

$$\dfrac{\dfrac{\underline{\mathtt{x}} \vdash \mathtt{P}}{\underline{\mathtt{x}}, \underline{\mathtt{y}} \vdash \mathtt{P}}\,W \qquad \dfrac{\underline{\mathtt{y}} \vdash \mathtt{Q}}{\underline{\mathtt{x}}, \underline{\mathtt{y}} \vdash \mathtt{Q}}\,W}{\underline{\mathtt{x}}, \underline{\mathtt{y}} \vdash \mathtt{PQ}}\,@$$

where the dashed rules are Weakenings, which are admissible from Lemma 1.3(2).

Case $\lambda\mathtt{x}.\mathtt{P}$: By inductive hypothesis we have $\underline{\mathtt{y}} \vdash \mathtt{P}$. Now we have two subcases[27]:

subcase $\mathtt{x} \in \underline{\mathtt{y}}$: then $\dfrac{\underline{\mathtt{y}} \vdash \mathtt{P}}{\underline{\mathtt{y}} - \{\mathtt{x}\} \vdash \lambda\mathtt{x}.\mathtt{P}}\lambda\mathtt{x}$

subcase $\mathtt{x} \notin \underline{\mathtt{y}}$: then

$$\dfrac{\dfrac{\underline{\mathtt{y}} \vdash \mathtt{P}}{\underline{\mathtt{y}}, \mathtt{x} \vdash \mathtt{P}}\,W}{\underline{\mathtt{y}} \vdash \lambda\mathtt{x}.\mathtt{P}}\,\lambda\mathtt{x}$$

□

Proof of Lemma 1.6. The equations for FV are immediate by definition of FV on nets (and in fact they hold for nets, in general).

That $FV(\mathtt{M})$ is contained in any context variable list for $\mathtt{M}$ is immediate by induction on $\mathtt{M} \in \Lambda$.

Let us now show, by induction on $\mathtt{M} \in \Lambda$, that $FV(\mathtt{M}) \vdash \mathtt{M}$. This is analogue to the previous proof just above.

Case $\mathtt{x}$: immediate.

Case $\mathtt{P}\,\mathtt{Q}$: We have the following derivation, where the top ones are given by the IH and the last rule uses the equations for FV shown above.

$$\dfrac{\dfrac{FV(\mathtt{P}) \vdash \mathtt{P}}{FV(\mathtt{P}), FV(\mathtt{Q}) \vdash \mathtt{P}}\,W \qquad \dfrac{FV(\mathtt{Q}) \vdash \mathtt{Q}}{FV(\mathtt{P}), FV(\mathtt{Q}) \vdash \mathtt{Q}}\,W}{FV(\mathtt{P}\,\mathtt{Q}) \vdash \mathtt{PQ}}\,@$$

Case $\lambda\mathtt{x}.\mathtt{P}$: By inductive hypothesis we have $FV(\mathtt{P}) \vdash \mathtt{P}$. Now we have two subcases[28]:

subcase $\mathtt{x} \in FV(\mathtt{P})$: then $\dfrac{FV(\mathtt{P}) \vdash \mathtt{P}}{FV(\mathtt{P}) - \{\mathtt{x}\} \vdash \lambda\mathtt{x}.\mathtt{P}}\lambda\mathtt{x}$ and $FV(\lambda\mathtt{x}.\mathtt{P}) = FV(\mathtt{P}) - \{\mathtt{x}\}$.

subcase $\mathtt{x} \notin FV(\mathtt{P})$: then

$$\dfrac{\dfrac{FV(\mathtt{P}) \vdash \mathtt{P}}{FV(\mathtt{P}), \mathtt{x} \vdash \mathtt{P}}\,W}{FV(\mathtt{P}) \vdash \lambda\mathtt{x}.\mathtt{P}}\,\lambda\mathtt{x}$$

and $FV(\lambda\mathtt{x}.\mathtt{P}) = FV(\mathtt{P}) - \{\mathtt{x}\} = FV(\mathtt{P})$. □

[27] Remark that we *cannot* use (α) in order to pick $\mathtt{x} \notin \underline{\mathtt{y}}$, since $\underline{\mathtt{y}}$ is not fixed independently from $\mathtt{x}$. In fact, it is the choice of $\mathtt{x}$ to be popped out that determines the actual $\mathtt{P}$, and $\underline{\mathtt{y}}$ is determined precisely from $\mathtt{P}$ (via the IH).

[28] Remark that, again, here we *cannot* use (α) in order to pick $\mathtt{x} \notin FV(\mathtt{P})$.

The interpretation is well-defined

Definition 2.1 is not really precise, for two reasons:

1) We have to check that the definition of the function does not depend on the chosen derivation. Otherwise the λ-abstraction is not well-defined, since we can pop any variable from it, and the resulting derivations are, strictly speaking, different. However, they only differ on the renaming of that variable, which in the definition only plays the role of placeholder. We have the impression that it is not a major issue: this is correct, but it requires some care.

2) We use $\lambda(\dots)$, which applies to functions $R \to R$. But we apply it on a function $R^{\mathtt{y}} \to R$.

Let us solve issue 2), as there is nothing serious going on and it is just a matter of abbreviations: writing $\lambda(f)$ for $f : R^{\mathtt{y}} \to R$, is sugar for $\lambda(\hat{f})$, where $\hat{f} : a \in R \mapsto f(\hat{a}) \in R$, where $\hat{a} \in R^{\mathtt{y}}$ is trivially defined by $\hat{a}_{\mathtt{y}} := a$.

Now let us turn to issue 1):

We can solve it in two ways: either we stipulate that, *a priori*, Definition 2.1 does not define *one* function, but instead a set of functions (yet to be precised). Once this properly defined, we show that this set is actually a singleton, so we obtain exactly one function indeed. Or, we first stipulate that we are not interpreting λ-term-with-context-variables, but we are interpreting the very derivations of a λ-term-with-context-variables (so, the trees whose root is a λ-term-with-context-variables). Then, we show that the interpretation is actually invariant with respect to derivations of the same λ-term-with-context-variables and we can therefore define, with no worries, the interpretation of one λ-term-with-context-variables as the interpretation of any of its derivations.

The mathematics of the two approaches is essentially the same. We choose to follow the second approach because more standard. Here it is:

Read Definition 2.1 as defining the interpretation of derivations, i.e.:

$$\llbracket \mathit{proj}^{\mathtt{x}}_{\underline{\mathtt{x}}} \rrbracket := \mathrm{proj}^{\mathtt{x}}_{\underline{\mathtt{x}}}, \quad \underline{a} \in R^{\underline{\mathtt{x}}} \mapsto a_x \in R \qquad\qquad \llbracket @(\pi, \pi') \rrbracket := \quad \underline{a} \in R^{\underline{\mathtt{x}}} \mapsto @(\llbracket \pi \rrbracket(\underline{a}), \llbracket \pi' \rrbracket(\underline{a})) \in R$$

$$\left\llbracket \frac{\pi : (\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{M})}{\lambda \mathtt{y}.\pi} \right\rrbracket := \quad \underline{a} \in R^{\underline{\mathtt{x}}} \mapsto \lambda\left(\llbracket \pi \rrbracket(\underline{\mathtt{x}} := \underline{a}, \mathtt{y} := (\_))\right) \in R$$

Notice that this is well-defined.

Remark that, reading the proofs of Lemma 1.3, Proposition 1.4 and Lemma 1.6 given in Section 3, we see that they actually *define* derivations witnessing the respective terms-with-context-variables. Here we only need the derivations defined in Lemma 1.3(1).

Definition 3.5. *The proof of Lemma 1.3(1) lifts any bijection $\sigma : \underline{\mathtt{x}} \xrightarrow{\simeq} \underline{\mathtt{z}}$ to a relation, which we call* σ-renaming *(still denoted σ), between derivations of the terms-with-context-variables $\underline{\mathtt{x}}$ and those with-context-variables $\underline{\mathtt{z}}$.*

Moreover, for all $\pi : (\underline{\mathtt{x}} \vdash \mathtt{M})$ and all σ with domain $\underline{\mathtt{x}}$, one has $\pi\sigma\rho$ iff $\rho : (\underline{\mathtt{z}} \vdash \mathtt{M}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\})$.

Proof that the definition makes sense and that it satisfies the claimed characterisation. The definition of σ-renaming is explicitly given below, by induction on π. At the same time, in the same induction, we also prove that, for all σ, σ-renaming satisfies the claimed characterisation.

Case $\dfrac{}{\underline{\mathtt{x}} \vdash \mathtt{x}}$: then for all σ,

$$\pi\sigma\rho \quad \textit{iff (by def)} \quad \rho = \frac{}{\underline{\mathtt{z}} \vdash \sigma(\mathtt{x})} \quad \textit{iff} \quad \rho : (\underline{\mathtt{z}} \vdash \mathtt{x}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}).$$

Case $\dfrac{\pi_1 : (\underline{\mathtt{x}} \vdash \mathtt{P}) \quad \pi_2 : (\underline{\mathtt{x}} \vdash \mathtt{Q})}{\underline{\mathtt{x}} \vdash \mathtt{P\,Q}}$: then for all σ,

$$\pi\sigma\rho \quad \textit{iff (by def)} \quad \rho = @(\rho_1, \rho_2) \quad \textit{with } \pi_1\sigma\rho_1 \textit{ and } \pi_1\sigma\rho_1$$

$$\textit{iff (by IH)} \quad \rho = \frac{\rho_1 : (\underline{\mathtt{z}} \vdash \mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}) \qquad \rho_2 : (\underline{\mathtt{z}} \vdash \mathtt{Q}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\})}{\underline{\mathtt{z}} \vdash (\mathtt{PQ})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}}$$

$$\textit{iff} \qquad \rho : (\underline{\mathtt{z}} \vdash (\mathtt{PQ})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\}).$$

Case $\dfrac{\pi' : (\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{P})}{\underline{\mathtt{x}} \vdash \lambda \mathtt{y}.\mathtt{P}}$, with $\mathtt{y} \notin \underline{\mathtt{x}}$: then for all σ,

$$\pi\sigma\rho \quad \textit{iff (by def)} \quad \rho = \lambda \mathtt{y}'.\rho' \qquad \begin{array}{l} \textit{with } \mathtt{y}' \notin FV(\mathtt{P}) \cup \underline{\mathtt{z}} \cup \underline{\mathtt{x}} \\ \textit{and } \pi'\widetilde{\sigma}\rho' \\ \textit{where } \widetilde{\sigma} := \sigma + [\mathtt{y} \mapsto \mathtt{y}'] \end{array}$$

$$\textit{iff (by IH)} \quad \rho = \frac{\rho' : (\underline{\mathtt{z}}, \mathtt{y}' \vdash \mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}}), \mathtt{y} := \mathtt{y}'\})}{\underline{\mathtt{z}} \vdash \lambda \mathtt{y}'.(\mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}}), \mathtt{y} := \mathtt{y}'\})}\lambda \mathtt{y}' \quad \textit{with } \mathtt{y}' \notin FV(\mathtt{P}) \cup \underline{\mathtt{z}} \cup \underline{\mathtt{x}}$$

$$\textit{iff} \qquad \rho : (\underline{\mathtt{z}} \vdash (\lambda \mathtt{y}.\mathtt{P})\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}})\})$$

The only non-immediate step is the last "iff", which hold thanks to Lemma 3.3. □

One can see that $\pi\sigma\pi'$ for some σ, iff one is obtained from the other by performing a renaming in any number of subderivations (including the whole derivation).

Remark 3.6. *From the def/prop one sees that $\pi, \pi' : (\underline{\mathtt{x}} \vdash \mathtt{M})$ iff $\pi\, \mathrm{id}_{\underline{\mathtt{x}}}\, \pi'$. This relation is called α-equivalence of derivations, and we denote it by $\pi =_\alpha \pi'$. The only difference with σ-renaming is that $=_\alpha$ corresponds to renaming any* proper *subderviation of a derivation.*

Remark that a bijection $\sigma : \underline{\mathtt{x}} \xrightarrow{\simeq} \underline{\mathtt{y}}$ lifts to a bijection $\sigma_R : R^{\underline{\mathtt{x}}} \to R^{\underline{\mathtt{y}}}$, defined by $\sigma_R(\underline{a})_y := a_{\sigma^{-1}(y)}$. Its inverse is given by $\sigma_R^{-1}(\underline{b})_x = b_{\sigma(x)}$.

Lemma 3.7. *1. Let $\sigma : \underline{\mathtt{x}} \xrightarrow{\simeq} \underline{\mathtt{z}}$ be a bijection. If $\pi\sigma\rho$, then $[\![\rho]\!] = [\![\pi]\!] \circ \sigma_R^{-1}$, i.e. $[\![\rho]\!]\,(\underline{\mathtt{z}} := \underline{b}) = [\![\pi]\!]\,(\underline{\mathtt{x}} := b_{\sigma(\underline{\mathtt{x}})})$.*

2. If $\pi =_\alpha \pi'$ then $[\![\pi]\!] = [\![\pi']\!]$.

Proof. (1). Induction on π.

Case $\dfrac{}{\underline{\mathtt{x}} \vdash \mathtt{x}}$: Then $\rho = \dfrac{}{\underline{\mathtt{z}} \vdash \sigma(\mathtt{x})}$ and $\left[\!\!\left[\dfrac{}{\underline{\mathtt{z}} := \underline{b} \vdash \sigma(\mathtt{x})}\right]\!\!\right] = b_{\sigma(x)} = \left[\!\!\left[\dfrac{}{\underline{\mathtt{x}} := b_{\sigma(\underline{\mathtt{x}})} \vdash \mathtt{x}}\right]\!\!\right]$.

Case $\dfrac{\pi_1 : (\underline{\mathtt{x}} \vdash \mathtt{P}) \quad \pi_2 : (\underline{\mathtt{x}} \vdash \mathtt{Q})}{\underline{\mathtt{x}} \vdash \mathtt{P\,Q}}$: Immediate by the IH (used twice).

Case $\dfrac{\pi' : (\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{P})}{\underline{\mathtt{x}} \vdash \lambda \mathtt{y}.\mathtt{P}}$, with $\mathtt{y} \notin \underline{\mathtt{x}}$: then $\rho = \dfrac{\rho' : (\underline{\mathtt{z}}, \mathtt{y}' \vdash \mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}}), \mathtt{y} := \mathtt{y}'\})}{\underline{\mathtt{z}} \vdash \lambda \mathtt{y}'.(\mathtt{P}\{\underline{\mathtt{x}} := \sigma(\underline{\mathtt{x}}), \mathtt{y} := \mathtt{y}'\})}$ with $\mathtt{y}' \notin FV(\mathtt{P}) \cup \underline{\mathtt{z}} \cup \underline{\mathtt{x}}$, for some ρ' such that $\pi'\widetilde{\sigma}\rho'$, where $\widetilde{\sigma} := \sigma + [\mathtt{y} \mapsto \mathtt{y}']$. Then $[\![\rho]\!]\,(\underline{\mathtt{z}} := \underline{b}) = \lambda([\![\rho']\!]\,(\underline{\mathtt{z}} := \underline{b}, \mathtt{y}' := (\_)))$ and $[\![\pi]\!]\,(\underline{\mathtt{x}} := b_{\sigma(\underline{\mathtt{x}})}) = \lambda([\![\pi']\!]\,(\underline{\mathtt{x}} := b_{\sigma(\underline{\mathtt{x}})}, \mathtt{y} := (\_)))$. Therefore, we are done if we show that

$$[\![\rho']\!]\,(\underline{\mathtt{z}} := \underline{b}, \mathtt{y}' := (\_)) = [\![\pi']\!]\,(\underline{\mathtt{x}} := b_{\sigma(\underline{\mathtt{x}})}, \mathtt{y} := (\_)) : R \to R.$$

But this holds, because for $c \in R$, the IH on $\pi'\widetilde{\sigma}\rho'$ gives $[\![\rho']\!]\,(\underline{\mathtt{z}} := \underline{b}, \mathtt{y}' := c) = [\![\pi']\!]\,(\underline{\mathtt{x}} := b_{\sigma(\underline{\mathtt{x}})}, \mathtt{y} := c)$, because $\widetilde{\sigma}_R^{-1} : R^{\underline{\mathtt{z}},\mathtt{y}'} \to R^{\underline{\mathtt{x}},\mathtt{y}}$ sends $(\underline{\mathtt{z}} := \underline{b}, \mathtt{y}' := c)$ to $(\underline{\mathtt{x}} := b_{\sigma(\underline{\mathtt{x}})}, \mathtt{y} := c)$.

(2). Immediate from (1) as $(\mathrm{id}_{\underline{\mathtt{x}}})_R^{-1} = \mathrm{id}_{R^{\underline{\mathtt{x}}}}$. □

Therefore, it makes sense to take the official definition of "$[\![\underline{\mathtt{x}} \vdash \mathtt{M}]\!]$" in Definition 2.1 as:

Definition 3.8. *The interpretation* $[\![\underline{\mathtt{x}} \vdash \mathtt{M}]\!]$ *of a* λ*-term-with-context-variables* $\underline{\mathtt{x}} \vdash \mathtt{M}$ *is the interpretation* $[\![\pi : (\underline{\mathtt{x}} \vdash \mathtt{M})]\!]$ *of any of its derivations. It is immediate to see that, with this definition, the equations in Definition 2.1 provide a characterization of the interpretation of terms-with-context-variables.*

Lemma 3.9. *Let* $\underline{\mathtt{x}} \vdash \mathtt{M}$*. If* $\mathtt{y} \notin \underline{\mathtt{x}}$*, then* $[\![\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{M}]\!] = [\![\underline{\mathtt{x}} \vdash \mathtt{M}]\!] \circ \mathrm{proj}^{\underline{\mathtt{x}},\mathtt{y}}_{\underline{\mathtt{x}}}$*, i.e.* $[\![\underline{\mathtt{x}}, \mathtt{y} \vdash \mathtt{M}]\!]\,(\underline{\mathtt{x}} := \underline{a}\,,\, \mathtt{y} := b) = [\![\underline{\mathtt{x}} \vdash \mathtt{M}]\!]\,(\underline{\mathtt{x}} := \ \underline{a})$*.*

Proof. Straightforward induction on $\underline{\mathtt{x}} \vdash \mathtt{M}$. □

Remark 3.10. *The reason why in ordinary* λ*-calculus we can forget about actual derivations, in the sense that those of the same term with contextual variables are all* α*-equivalent, is due to the fact that the derivations are "syntax directed", i.e. the kind of rule that we can apply only depends on the shape of the* λ*-term in the conclusion of the derivation. For other* λ*-calculi, especially for those that formalise non-trivial notions of proof*[29]*, this may not be the case (there may be two different proofs of the same judgment which differ not only in logically meaningless aspects such as renaming of variables, but really logically different, i.e. using different rules but deriving the same judgment). In such cases one has to deal with derivations and not just with the conclusion judgment, while still taking them modulo* α*-equivalence (once properly defined), since those are syntactic non-logical/non-computational differences (in practice, one manipulates derivations* before α*-equivalence, and makes renaming on the fly whenever needed). Then one has to make sure all the constructions are compatible with* α*-equivalence (especially the cut-equivalence, which in our case would be* $(=_\beta)$*; for this, one typically needs to use e.g. a weakening rule; however, one cannot only use the fact that if a judgment is provable then its weakening is, but needs to define the weakening of a derivation modulo* α*-equivalence).*

Proof of Theorem 2.5

Theorem 3.11. *Let* $\underline{\mathtt{y}}, \mathtt{x} \vdash \mathtt{M}$ *with* $\mathtt{x} \notin \underline{\mathtt{y}}$*, and* $\underline{\mathtt{y}} \vdash \mathtt{N}$*. Then:*

$$\begin{aligned}\Big[\!\!\Big[\underline{\mathtt{y}} \vdash (\lambda \mathtt{x}.\mathtt{M})\mathtt{N}\Big]\!\!\Big] = \Big[\!\!\Big[\underline{\mathtt{y}} \vdash \mathtt{M}\{\mathtt{x} := \mathtt{N}\}\Big]\!\!\Big] : \quad & R^{\underline{\mathtt{y}}} \to R \\ & \underline{b} \mapsto \Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b}, \mathtt{x} := \Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N}\Big]\!\!\Big] \vdash \mathtt{M}\Big]\!\!\Big]\end{aligned} \qquad ([\![\beta]\!])$$

Proof. Developing the definitions we immediately see that

$$\begin{aligned}\Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b} \vdash (\lambda \mathtt{x}.\mathtt{M})\mathtt{N}\Big]\!\!\Big] &= @\Big(\Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b} \vdash \lambda \mathtt{x}.\mathtt{M}\Big]\!\!\Big]\,,\,\Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N}\Big]\!\!\Big]\Big) \\ &= @\Big(\lambda\Big(\Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b}, \mathtt{x} := (\_) \vdash \mathtt{M}\Big]\!\!\Big]\Big)\,,\,\Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N}\Big]\!\!\Big]\Big) \\ &= \Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b}, \mathtt{x} := \Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N}\Big]\!\!\Big] \vdash \mathtt{M}\Big]\!\!\Big] \qquad \textit{by } (\beta).\end{aligned}$$

Therefore, we are done if we show that $\Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b} \vdash \mathtt{M}\{\mathtt{x} := \mathtt{N}\}\Big]\!\!\Big] = \Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b}, \mathtt{x} := \Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N}\Big]\!\!\Big] \vdash \mathtt{M}\Big]\!\!\Big]$ for all $\underline{\mathtt{y}}, \mathtt{x} \vdash \mathtt{M}$, $\underline{\mathtt{y}} \vdash \mathtt{N}$, $\mathtt{x} \notin \underline{\mathtt{y}}$ and $\underline{b} \in R^{\underline{\mathtt{y}}}$. We do this by induction on $\underline{\mathtt{y}}, \mathtt{x} \vdash \mathtt{M}$.

Case $\dfrac{}{\underline{\mathtt{y}}, \mathtt{x} \vdash \mathtt{x}}$:

then the LHS is $\Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N}\Big]\!\!\Big]$, and the RHS is $\mathrm{proj}^{\underline{\mathtt{y}},\mathtt{x}}_{\mathtt{x}}\Big(\underline{\mathtt{y}} := \underline{b}\,,\, \mathtt{x} := \Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N}\Big]\!\!\Big]\Big) = \Big[\!\!\Big[\underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N}\Big]\!\!\Big]$.

Case $\dfrac{}{\underline{\mathtt{y}}, \mathtt{x} \vdash \mathtt{y}}$:

[29] But not the ones that we will see in Lecture 3, which rely on the ordinary λ-calculus.

then the LHS is $\mathrm{proj}^{\underline{\mathtt{y}}}_{\mathtt{y}}\left(\underline{\mathtt{y}} := \underline{b}\right) = b_y$ and the RHS is $\mathrm{proj}^{\underline{\mathtt{y}},\mathtt{x}}_{\mathtt{y}}\left(\underline{\mathtt{y}} := \underline{b}\,,\, \mathtt{x} := \llbracket \underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N} \rrbracket\right) = b_y$.

Case $\dfrac{\underline{\mathtt{y}}, \mathtt{x} \vdash \mathtt{P} \quad \underline{\mathtt{y}}, \mathtt{x} \vdash \mathtt{Q}}{\underline{\mathtt{y}}, \mathtt{x} \vdash \mathtt{P}\,\mathtt{Q}}$: Then the LHS is

$$\begin{array}{rcl}
\llbracket \underline{\mathtt{y}} := \underline{b} \vdash (\mathtt{PQ})\{\underline{\mathtt{x}} := \mathtt{N}\} \rrbracket & = & @\left(\llbracket \underline{\mathtt{y}} := \underline{b} \vdash \mathtt{P}\{\underline{\mathtt{x}} := \mathtt{N}\} \rrbracket\,,\, \llbracket \underline{\mathtt{y}} := \underline{b} \vdash \mathtt{Q}\{\underline{\mathtt{x}} := \mathtt{N}\} \rrbracket\right) \\
& = & @\left(\llbracket \underline{\mathtt{y}} := \underline{b}, \mathtt{x} := \llbracket \underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N} \rrbracket \vdash \mathtt{P} \rrbracket\,,\, \llbracket \underline{\mathtt{y}} := \underline{b}, \mathtt{x} := \llbracket \underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N} \rrbracket \vdash \mathtt{Q} \rrbracket\right) \\
& = & \llbracket \underline{\mathtt{y}} := \underline{b}, \mathtt{x} := \llbracket \underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N} \rrbracket \vdash \mathtt{PQ} \rrbracket
\end{array}$$

which is the RHS.

Case $\dfrac{\underline{\mathtt{y}}, \mathtt{x}, \mathtt{z} \vdash \mathtt{P}}{\underline{\mathtt{y}}, \mathtt{x} \vdash \lambda \mathtt{z}.\mathtt{P}}$ with $\mathtt{z} \notin \underline{\mathtt{y}} \cup \{\mathtt{x}\}$: Wlog by Remark 3.4, $\mathtt{z} \notin \underline{\mathtt{y}} \cup \{\mathtt{x}\} \cup FV(\mathtt{N})$. Then (as $\mathtt{z} \notin FV(\mathtt{N})$) the LHS is

$$\begin{array}{rcl}
\llbracket \underline{\mathtt{y}} := \underline{b} \vdash \lambda \mathtt{z}.(\mathtt{P}\{\mathtt{x} := \mathtt{N}\}) \rrbracket & = & \lambda(\llbracket \underline{\mathtt{y}} := \underline{b}, \mathtt{z} := (\_) \vdash \mathtt{P}\{\mathtt{x} := \mathtt{N}\} \rrbracket) \\
& = & \lambda(\llbracket \underline{\mathtt{y}} := \underline{b}, \mathtt{z} := (\_), \mathtt{x} := \llbracket \underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N} \rrbracket \vdash \mathtt{P} \rrbracket) \\
& = & \llbracket \underline{\mathtt{y}} := \underline{b}, \mathtt{x} := \llbracket \underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N} \rrbracket \vdash \lambda \mathtt{z}.\mathtt{P} \rrbracket
\end{array}$$

which is the RHS. The only non-immediate step is the second equality, which follows because the two functions under the λ are the same function: for $c \in R$, the IH is applicable to $\underline{\mathtt{y}}, \mathtt{x}, \mathtt{z} \vdash \mathtt{P}$ because $\mathtt{x} \notin \underline{\mathtt{y}} \cup \{\mathtt{z}\}$, and gives $\llbracket \underline{\mathtt{y}} := \underline{b}, \mathtt{z} := c \vdash \mathtt{P}\{\mathtt{x} := \mathtt{N}\} \rrbracket = \llbracket \underline{\mathtt{y}} := \underline{b}, \mathtt{z} := c, \mathtt{x} := \llbracket \underline{\mathtt{y}} := \underline{b} \vdash \mathtt{N} \rrbracket \vdash \mathtt{P} \rrbracket$. $\square$

4 Solution to Exercises

Answer (Ex. 1) — By following Proposition ?? of Lecture 1, one can see that $\llbracket \mathtt{f} \vdash \widetilde{\mathtt{Y}} \rrbracket = Y \circ \mathit{fun}$, where $\widetilde{\mathtt{Y}} := (\lambda \mathtt{x}.\mathtt{f}(\mathtt{x}\,\mathtt{x}))(\lambda \mathtt{x}.\mathtt{f}(\mathtt{x}\,\mathtt{x}))$. Setting $\mathtt{Y} := \lambda \mathtt{f}.\widetilde{\mathtt{Y}}$ we have thus $\llbracket \vdash \mathtt{Y} \rrbracket = \lambda(Y \circ \mathit{fun})$, and the result follows from Proposition 2.6. $\square$

Answer (Ex. 3) — Suppose that $\mathtt{M} \in \Lambda$ has two normal forms $\mathtt{N}_1, \mathtt{N}_2 \in \Lambda$. That is, $\mathtt{N}_1, \mathtt{N}_2$ are normal and we have $\mathtt{N}_1 \twoheadleftarrow \mathtt{M} \twoheadrightarrow \mathtt{N}_2$. By confluence there is $\mathtt{H} \in \Lambda$ such that $\mathtt{N}_1 \twoheadrightarrow \mathtt{H} \twoheadleftarrow \mathtt{N}_2$. But since $\mathtt{N}_1, \mathtt{N}_2$ are normal, the only $\twoheadrightarrow$-reduction from them can be a 0-step reduction, so $\mathtt{N}_1 = \mathtt{H} = \mathtt{N}_2$. $\square$

Answer (Ex. 5) — For *not*, we look for a term NOT with the following specification:

$$\begin{cases} \text{NOT TRUE} =_\beta \text{FALSE} \\ \text{NOT FALSE} =_\beta \text{TRUE} \end{cases}$$

One easy choice is:

$$\text{NOT} := \lambda \mathtt{b}.\,\mathtt{b}\,\text{FALSE TRUE}.$$

Remark that, even if it doesn't precisely fit in the above definition, we can implement the *if-then-else*, in the sense that we look for a term IFTE with the following specification:

$$\begin{cases} \text{IFTE TRUE}\,\mathtt{M}\,\mathtt{N} =_\beta \mathtt{M} \\ \text{IFTE FALSE}\,\mathtt{M}\,\mathtt{N} =_\beta \mathtt{N} \end{cases}$$

This is left as an exercise to the reader.
For *and*, we are back to the definition given in the text, and we look for a term AND with the following specification, given by the truth table of the *and*:

$$\begin{cases} \text{AND TRUE TRUE} =_\beta \text{TRUE} \\ \text{AND TRUE FALSE} =_\beta \text{FALSE} \\ \text{AND FALSE TRUE} =_\beta \text{FALSE} \\ \text{AND FALSE FALSE} =_\beta \text{FALSE} \end{cases}$$

One possible choice, whose idea consists in inspecting the first argument, is

$$\text{AND} := \lambda \mathtt{m}\,\mathtt{n}.\,\mathtt{m}\,(\mathtt{n}\,\text{TRUE FALSE})\,\text{FALSE}.$$

One can also inspect the second argument (which corresponds to the fact that AND is total and symmetric), yielding

$$\text{AND}' := \lambda \mathtt{m}\,\mathtt{n}.\,\mathtt{n}\,(\mathtt{m}\,\text{TRUE FALSE})\,\text{FALSE}.$$

There is also another smarter implementation::

$$\text{AND}''' = \lambda \mathtt{m}\,\mathtt{n}.\,\mathtt{m}\,\mathtt{n}\,\mathtt{m}.$$

Check that it works.
Similarly, for *or* we have

$$\text{OR} := \lambda \mathtt{m}\,\mathtt{n}.\,\mathtt{m}\,\text{TRUE}\,(\mathtt{n}\,\text{TRUE FALSE}) \text{ *or also* } \text{OR}' := \lambda \mathtt{m}\,\mathtt{n}.\,\mathtt{n}\,\text{TRUE}\,(\mathtt{m}\,\text{TRUE FALSE}) \text{ *or also* } \text{OR}'' := \lambda \mathtt{m}\,\mathtt{n}.\,\mathtt{m}\,\mathtt{m}\,\mathtt{n}.$$

For *xor*, we have

$$\text{XOR} := \lambda \mathtt{m}\,\mathtt{n}.\,\mathtt{m}\,(\mathtt{n}\,\text{FALSE TRUE})\,(\mathtt{n}\,\text{TRUE FALSE}) \text{ *or also* } \text{XOR}' := \lambda \mathtt{m}\,\mathtt{n}.\,\mathtt{n}\,(\mathtt{m}\,\text{FALSE TRUE})\,(\mathtt{m}\,\text{TRUE FALSE}).$$

Of course one can also combine AND, OR, NOT, XOR in order to define them in terms of each other. □

Answer (Ex. 6) —

$$\text{SUCC} := \lambda \mathtt{n}\,\mathtt{f}\,\mathtt{x}.\,\mathtt{f}(\mathtt{n}\,\mathtt{f}\,\mathtt{x})$$

Another possibility is $\text{SUCC} := \lambda \mathtt{n}\,\mathtt{f}\,\mathtt{x}.\,\mathtt{n}\,\mathtt{f}\,(\mathtt{f}\,\mathtt{x})$.

$$\text{ADD} := \lambda \mathtt{n}\,\mathtt{m}\,\mathtt{f}\,\mathtt{x}.\,\mathtt{n}\,\mathtt{f}\,(\mathtt{m}\,\mathtt{f}\,\mathtt{x})$$

Another possibility is $\text{ADD} := \lambda \mathtt{n}\,\mathtt{m}\,\mathtt{f}\,\mathtt{x}.\,\mathtt{m}\,\mathtt{f}\,(\mathtt{n}\,\mathtt{f}\,\mathtt{x})$.

$$\text{ISZERO} := \lambda \mathtt{n}.\,\mathtt{n}\,(\lambda\_.\,\text{FALSE})\,\text{TRUE}$$

It is immediate to prove all the above satisfy the required specifications. □

Answer (Ex. 7) —

$$\text{SUBTR} := \lambda \mathtt{n}\,\mathtt{m}.\,\mathtt{m}\,\text{PRED}\,\mathtt{n}$$

$$\text{LEQ} := \lambda \mathtt{n}\,\mathtt{m}.\,\text{ISZERO}\,(\text{SUBTR}\,\mathtt{n}\,\mathtt{m})\,\text{TRUE FALSE}$$

$$\text{EQUALS} := \lambda \mathtt{n}\,\mathtt{m}.\,\text{AND}\,(\text{LEQ}\,\mathtt{n}\,\mathtt{m})\,(\text{LEQ}\,\mathtt{m}\,\mathtt{n})$$

$$\text{LESS} := \lambda \mathtt{n}\,\mathtt{m}.\,\text{AND}\,(\text{LEQ}\,\mathtt{n}\,\mathtt{m})\,(\text{NOT}\,(\text{EQUALS}\,\mathtt{n}\,\mathtt{m}))$$

It is immediate to prove all the above satisfy the required specifications. □

Answer (Ex. 8) — 1. We show i) and ii) of the hint to the exercise.
i)
Suppose that q, r are the Euclidean division functions (in particular, they are total on $\mathbb{N} \times \mathbb{N}_{>0}$), and let us show that they satisfy the equations of the exercise.
Interestingly, we do not need any induction[30], and a simple reasoning by cases on $m \geq 0$ suffices.
Case 0: We have $0 = n \cdot q(0, n) + r(0, n)$. Since all those numbers are non-negative, it must be $r(0, n) = 0$ and $n \cdot q(0, n) = 0$. Since $n \neq 0$, the latter entails $q(0, n) = 0$.
Case $m + 1$: Let us consider $m + 1 = n \cdot q(m + 1, n) + r(m + 1, n)$ and $m = n \cdot q(m, n) + r(m, n)$, with $r(m + 1, n)$, $r(m, n) < n$. Substituting the second equation in the first, we get

$$n \cdot (q(m, n) - q(m + 1, n)) + r(m, n) + 1 = r(m + 1, n). \qquad (\star)$$

Remark that since $r(m, n) < n$, then $r(m, n) + 1 \leq n$. So we have two subcases:
Subcase $r(m, n)+1 < n$. Then $r(m, n)+1 = r(m, n)+1 \bmod n$. Therefore, since also $r(m+1, n) < n$, taking the $\bmod n$ of $(\star)$ we get $r(m + 1, n) = r(m, n) + 1$, which is one desired equality. Substituting it back in $(\star)$, we also get $n \cdot (q(m, n) - q(m + 1, n)) = 0$ and, since $n \neq 0$, we have $q(m + 1, n) = q(m, n)$, which is another desired equality.
Subcase $r(m, n) + 1 = n$. Substituting this in the $(\star)$, we have $n \cdot (q(m, n) - q(m + 1, n) + 1) = r(m + 1, n)$. But $r(m, n) < n$, so $r(m, n) = r(m, n) \bmod n$, and by taking $\bmod n$ of the latter equation we get $r(m + 1, n) = 0$, which is yet another desired equality. Substituting it back in $(\star)$ and using $r(m + 1, n) + 1 = n$, we get $n \cdot (q(m, n) - q(m + 1, n) + 1) = 0$ and, since $n \neq 0$, we have $q(m + 1, n) = q(m, n) + 1$, which is the last desired equality.
ii)
Suppose that q, r are defined by the equations of the exercise and let us show that they compute the Euclidean division (in particular, they are toatl on $\mathbb{N} \times \mathbb{N}_{>0}$).
We first show that they are total, and then[31] that they compute the Euclidean division.
For the totality (on $\mathbb{N} \times \mathbb{N}_{>0}$), we see that, by definition, q is total iff r is, so it suffices to show that r is total. We see from the definition of r that the only possibility for $r(m, n)$ not to be defined is if $m = m'+1$ and either $r(m', n)$ is not defined, or it is and $r(m', n)+1 > n$. Therefore, the totality of r follows from ruling out the above possibility, as in the following claim: *For all $m \geq 0$ and $n > 0$, $r(m, n)$ is defined and $r(m, n) + 1 \leq n$.*
We show the claim by induction on m: Case 0: $r(0, n)$ is defined to be 0 and $1 \leq n$. Case $m + 1$: By IH, $r(m, n)$ is defined and $r(m, n) + 1 \leq n$. So $r(m + 1)$ is defined and it is either 0, if $r(m, n)+1 = n$, or $r(m, n)+1$, if $r(m, n)+1 < n$. In the first case we have $r(m+1, n)+1 = 1 \leq n$. In the second case we have $r(m + 1, n) + 1 = r(m, n) + 2 < n + 1$, so $r(m + 1, n) + 1 \leq n$.
Now the we know that they are total, let us show that q, r compute the Euclidean division, i.e. they satisfy $m = n \cdot q(m, n) + r(m, n)$ and $r(m, n) < n$.
Remark that from the claim above, we already get that $r(m, n) < n$. So we only prove the remaining equality. We do this by induction on m:
Case 0: we need to show that $0 = n \cdot 0 + 0$, which is trivial.
Case $m + 1$: we need to show that $m + 1 = n \cdot q(m + 1, n) + r(m + 1, n)$. By IH we have $m = n \cdot q(m, n) + r(m, n)$. Now, following the definition of q, r, we have the following two subcases (we know that q, r are total, so those are the only possibilities):
Subcase $r(m, n) + 1 < n$: then using the IH and the definition of q, r, we have the desired: $m + 1 = n \cdot q(m, n) + r(m, n) + 1 = n \cdot q(m + 1, n) + r(m + 1, n)$.
Subcase $r(m, n)+1 = n$: then using $r(m, n)+1 = n$, the IH and the definition of q, r we have the desired: $m+1 = n \cdot q(m, n) + r(m, n) + 1 = n \cdot q(m, n) + n = n \cdot (q(m, n) + 1) = n \cdot q(m + 1, n) = n \cdot q(m + 1, n) + r(m + 1, n)$.

[30] This is because we already suppose that they are total.
[31] We could do both steps together, but for the sake of clarity let us distinguish them.

2. We need to put q, r in the shape of recursive functions with g, h as in Theorem 2.22. It is immediate to see that we can take $g_r = g_q = 0$. However, remark that, even if q, r are total, a naif inspection of their definition could let us define h_r, h_q by cases as in the equations in the exercise, e.g. $h_r(a, b) = a + 1$ if $a + 1 < b$ and $= 0$ if $a + 1 = b$. But those are not total, and so we cannot apply Theorem 2.22. However this is not an issue[32]: we can make h_r, h_q total by defining them arbitrarily in the case $a + 1 > b$, since this case will never be taken in q, r. In fact, it is immediate to see that the following total h_r, h_q satisfy: $r(m + 1, n) = h_r(r(m, n), m, n)$ and $q(m + 1, n) = h_q(q(m, n), m, n)$.

$$h_r(a, d, b) := \begin{cases} a+1 & \textit{if } a+1 < b \\ 0 & \textit{if } a+1 = b \\ 42 & \textit{if } a+1 > b \end{cases} \qquad h_q(a, c, b) := \begin{cases} a & \textit{if } r(c, b)+1 < b \\ a+1 & \textit{if } r(c, b)+1 = b \\ 42 & \textit{if } r(c, b)+1 > b \end{cases}$$

We see that in order to implement h_q (and, then, q) we need to implement r first.
This is immediate to do by first implementing h_r and, then, writing the term which implement r using Theorem 2.22. But actually, since h_r does not use its second argument, we can equally well consider its slightly simpler variant $\widetilde{h}_r(a, b)$, and its implementation is:

$$\begin{array}{llll} \textsc{H}_r := \lambda \mathtt{a}\,\mathtt{b}. & \textsc{ifte}\,(\textsc{less}\,(\textsc{succ}\,\mathtt{a})\,\mathtt{b}) & & \\ & & (\textsc{succ}\,\mathtt{a}) & \\ & & (\textsc{ifte}\,(\textsc{equals}\,(\textsc{succ}\,\mathtt{a})\,\mathtt{b}) & \\ & & & \ulcorner 0 \urcorner \\ & & & \ulcorner 42 \urcorner \\ & &) & \end{array}$$

Now as in Theorem 2.22 we have that the Euclidean division reminder function r is implemented by the (slightly clearer) term (it is actually $=_\beta$ to the one obtained with $h_r(a, d, b)$):

$$\textsc{reminder} := \mathtt{Y}\,(\lambda \mathtt{r}\,\mathtt{m}\,\mathtt{n}.\,\textsc{isZero}\,\mathtt{m}\,\ulcorner 0 \urcorner\,(\textsc{H}_r\,(\mathtt{r}\,(\textsc{pred}\,\mathtt{m})\,\mathtt{n})\,\mathtt{n})).$$

Let us now conclude with the quotient: we can immediately implement h_q as:

$$\begin{array}{lll} \textsc{H}_q := \lambda \mathtt{a}\,\mathtt{c}\,\mathtt{b}.\,\textsc{ifte}\,(\textsc{less}\,(\textsc{succ}\,(\textsc{reminder}\,\mathtt{c}\,\mathtt{b}))\,\mathtt{b}) & & \\ & \mathtt{a} & \\ & (\textsc{ifte}\,(\textsc{equals}\,(\textsc{succ}\,(\textsc{reminder}\,\mathtt{c}\,\mathtt{b}))\,\mathtt{b}) & \\ & & (\textsc{succ}\,\mathtt{a}) \\ & & \ulcorner 42 \urcorner \\ &) & \end{array}$$

From this, Theorem 2.22 says that the Euclidean division quotient function is implemented by:

$$\textsc{quotient} := \mathtt{Y}\,(\lambda \mathtt{q}\,\mathtt{m}\,\mathtt{n}.\,\textsc{isZero}\,\mathtt{m}\,\ulcorner 0 \urcorner\,(\textsc{H}_q\,(\mathtt{q}\,(\textsc{pred}\,\mathtt{m})\,\mathtt{n})\,(\textsc{reminder}\,(\textsc{pred}\,\mathtt{m})\,\mathtt{n})\,\mathtt{n})).$$

Remark that since h_r and h_q are defined arbitrarily if $a + 1 > b$ and $c + 1 > b$, in that part of $\textsc{H}_q, \textsc{H}_r$ we could have actually put any λ-term: since we know that q, r are total, we would still have obtained an implementation of them[33].

[32] Actually, in order to represent q, r in λ-calculus there is not even need to make them total: as we discussed in Theorem 2.23, λ-calculus implements all partial recursive functions! But since we did not discuss how to deal with partiality (see next footnote), and we proved Theorem 2.22 only for total functions, we make them total in this exercise. Remark that the fact that, here, we could make them total by arbitrarily defining them, is a special property of q, r. It is well-known, that in general partial functions are needed.

[33] In particular, we could have put Ω instead of $\ulcorner 42 \urcorner$. This correspond to keeping h_r, h_q not defined in that case, and it is indeed how one deals with partiality...

Finally, remark how the above implementations are absolutely disastrous from the point of view of complexity. Of course one can optimise the code, but here we were just concerned with the fact that an implementation exists.

□

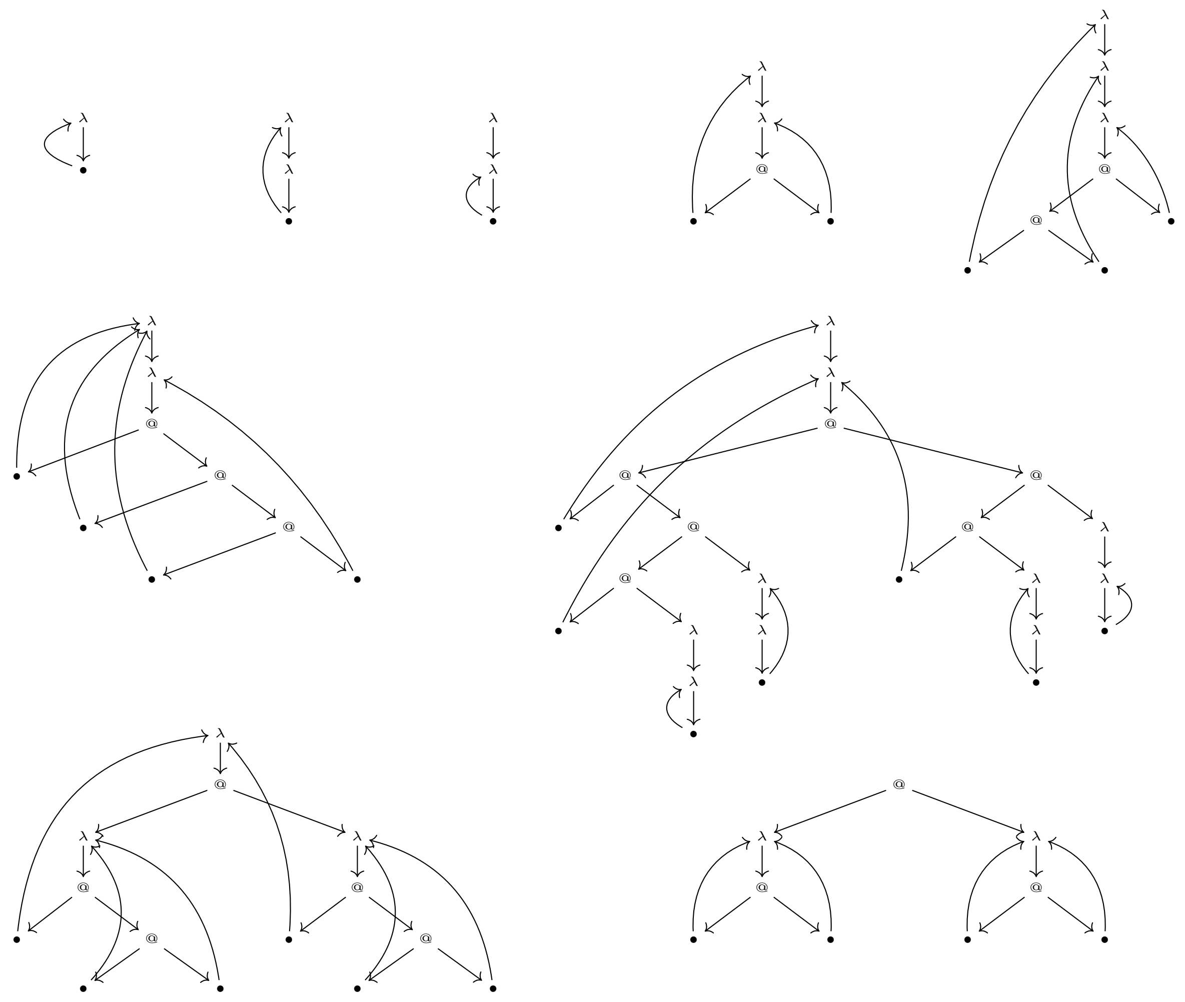


Figure 1: Some (closed) λ-terms. From left to right (numbers and booleans are in Church encoding):
First line, $\mathtt{I}, \textsc{true}, \textsc{false}, \mathtt{1}, \mathtt{ifte}$; Second line, $\mathtt{3}, \mathtt{xor}$; Third line, $\mathtt{Y}, \mathtt{\Omega}$.
Of course programming with these objects would be impracticable. Therefore the actual representation of λ-terms that we have chosen (in terms of such nets) has to be seen as machine code. Writing them as terms in the grammar of Definition 1.4 – in addition to being, strictly speaking, their definition proving that a certain net is indeed a λ-term – is a sort of assembly language for it: it is clearer to humans and it abbreviates the machine code, while still becoming impracticable for non trivial programming. Starting from that, one could put types, take built-in data structures (or just encode them) and so on, so to have a relatively manageable PL. Taking this idea seriously brings to theoretical languages of fundamental importance, like MLLTs, F, PCF, but also, after due non-trivial adaptations, to some important real life PL, such as Haskell, Agda, OCaml, Rocq, Lean etc. Note that chosing an actual machine implementation of λ-calculus is not obvious, and lot of research is done for that (see nominal sets, high order abstract syntax, de brujin indices, etc). The choice that we took here is suggestive, but it is only one of the many, and anyway λ-terms are not actually implemented as those kind of nets in real life PL: what matters is their inductive nature, not the implementation that allows it.