Exercise 1 *(distinct terms of same type) The type $\sigma = \tau \to \tau \to \tau$ is per convention read as $\tau \to (\tau \to \tau)$. Setting $M_1 = \lambda xy.x$, $M_2 = \lambda xy.y$, and $M_3 = (\lambda z.z)\lambda xy.x$ we easily verify that, for $i \in \{1,2,3\}$, we have $\vdash M_i : \tau \to \tau \to \tau$ for an arbitrary type τ:*

$$\dfrac{\dfrac{x:\tau, y:\tau \vdash x:\tau}{x:\tau \vdash \lambda y.x : \tau \to \tau}}{\vdash \lambda xy.x : \tau \to \tau \to \tau} \qquad \dfrac{\dfrac{x:\tau, y:\tau \vdash y:\tau}{x:\tau \vdash \lambda y.y : \tau \to \tau}}{\vdash \lambda xy.y : \tau \to \tau \to \tau} \qquad \dfrac{\dfrac{z:\sigma \vdash z:\sigma}{\vdash \lambda z.z : \sigma \to \sigma} \quad \dfrac{\dfrac{x:\tau, y:\tau \vdash x:\tau}{x:\tau \vdash \lambda y.x : \tau \to \tau}}{\vdash \lambda xy.x : \tau \to \tau \to \tau}}{\vdash (\lambda z.z)\lambda xy.x : \tau \to \tau \to \tau}$$

What way to count is reasonable depends on one's perspective. As answer to the question whether a type is inhabited, each of them is as good as the other (yielding the answer 'yes'), so they can be simply identified. (Or via the Curry–Howard isomorphism 'a proof is a proof'.)[1] *However, viewing λ-terms as representing mathematical functions, i.e. sets of input–output pairs, M_1 and M_3 (having the first of its two inputs as output) can be distinguished from M_2 (returning the second of its two inputs). More precisely, λ-terms can be thought of as functional programs (in some very basic functional programming language) and programs being not $=_\beta$-related corresponds to them implementing distinct algorithms. From that perspective it is reasonable to count such as distinct. Finally, whether or not to count λ-terms that are $=_\beta$-related (note that $M_3 \to_\beta M_1$ in the above) as distinct, is up for debate: on the one hand they correspond to the same algorithm (indeed, many compilers would automatically 'inline' M_3 to M_1), on the other hand abstracting repeated code into separate methods/procedure may be considered good programming/proof practice.*[2]

Remark 1 *Most people gave λ-terms to which the type σ supposedly could be assigned. Usually these assertions were correct, but in principle they should be proven by giving some inference, as in the solution above. That would have prevented some incorrect answers, like $\lambda x.f$: although this can be assigned type $\tau \to \tau \to \tau$ that can only be done* under the assumption that *$f : \tau \to \tau$. That is, we would need $f : \tau \to \tau$ in the context, violating that the context be empty as specified. More generally, note that $\vdash M : \sigma$, i.e. that M may be assigned type σ in the* empty *context, entails that M is a term having* no free variables *(corresponding, via the Curry–Howard isomorphism, to that we can* prove *σ* without *assumptions).*

One may use Haskell for verifying types. For the M_i in the above exercise it gives the types:

```
Prelude> :t (\x y -> x)
```

[1] It is also up for debate what proofs are, what it means for two proofs to be 'the same', or whether one proof can be 'better than' another proof, and therefore how to count proofs. Via the Curry–Howard isomorphism the considerations for λ-terms also pertain to ND proofs. (Cf. e.g. the notion of proof irrelevance in the proof assistant Coq.)

[2] The question when to view programs as the same could also be a legal one e.g. for deciding plagiarism: do we consider two programs the same if only variables have been renamed ($M =_\alpha N$) ? if only some code has been abstracted into methods/procedures ($M =_\beta N$)? if they have the same complexity ($O(M) = O(N)$; merge sort vs. heap sort)?

```
(\x y -> x) :: p1 -> p2 -> p1
Prelude> :t (\x y -> y)
(\x y -> y) :: p1 -> p2 -> p2
Prelude> :t (\z -> z)(\x y -> x)
(\z -> z)(\x y -> x) :: p1 -> p2 -> p1
```

Note that Haskell returns the most general *type an expression can have, its so-called* principal *type, and that that is in each case here more general than the type we have. But the only thing relevant here is that we may arbitrarily instantiate* `p1` *and* `p2` *in the above, in particular we may instantiate both with* τ*. That is, there may be many types that can be assigned to a* λ*-term, but if a type can be assigned at all, there is a most general such, its principal type.*

Exercise 2 *(two the same among three) How many (closed)* λ*-terms in* normal form *are there having type* $\tau \to \tau \to \tau$ *for* τ *arbitrary? We claim there are exactly two such, when we interpret* τ *being arbitrary as knowing nothing about the structure of* τ*. This we simulate by assuming* τ *is a base type. By the subformula property we know that the only types occurring in a type inference of such terms are subtypes of the given type, i.e. either* τ *or* $\tau \to \tau$*. By the inhabitation-procedure given, this means that*

1. *such a term must have shape* $\lambda x.M$ *(since* $z\vec{Q}$ *is impossible as the variable* z *would be free, contradicting closedness). Then* x *will be assigned type* τ *and* M *assigned type* $\tau \to \tau$*;*

2. *the term* M *must have shape* $\lambda y.N$ *(since* $z\vec{Q}$ *is impossible as the variable* z *would either be free, contradicting closedness, or otherwise be equal to* x *but then* $M = x\vec{Q}$ *would have a type smaller than* $\tau \to \tau$ *contradicting that it has type* $\tau \to \tau$*). Then* y *will be assigned type* τ *and* N *assigned type* τ*.*

3. *the term* N *being of, per assumption, base type* τ *it is not of shape* $\lambda z.P$ *so must be of shape* $z\vec{Q}$*. By closedness* z *must be either* x *or* y*, and them both being of type* τ*,* $\vec{Q}$ *must be the empty vector.*

From the above, we conclude that there are only two λ*-terms in normal form of the given type, namely* $\lambda xy.x$ *and* $\lambda xy.y$*. (This corresponds via the Curry–Howard isomorphism to that the proposition* $p \supset p \supset p$ *only has two proofs: the first using the first assumption, named* x*, to conclude and the other using the second assumption, named* y *to do so.)*

Since the typed λ*-calculus is terminating (SN) every* λ*-term of that type must be convertible to one of these two terms, from which we conclude as desired.*

Remark 2 *Most people argued that among the* λ*-terms provided by them in Exercise 1, two* are $=_\beta$*-related. However, the exercise was to show that they* must be *so related, i.e. to show this holds irrespective of a particular choice.*

If we do know something about the structure of τ*, we may be able to construct more terms. For instance, if* $\tau = \sigma \to \sigma$*, then also* $\lambda xyz.z$ *can be assigned the given type.*

There is an algorithm to count the number of inhabitants of a type (returning either some non-negative number or infinity) due to Ben Yelles.

Although irrelevant for this exercise, note that $\lambda xy.x$ and $\lambda xy.y$ are not β-convertible, that is, not related by $=_\beta$. This follows from the Church–Rosser property/confluence, saying that if $M =_\beta N$, then M and N both reduce by $\to_\beta$ into some term Z, a so-called common reduct. But since both terms are normal forms, they do not reduce at all, so definitely they do not have a common reduct, so they cannot be $=_\beta$-related.

Exercise 3 *(reducing $\omega 1$) We indeed have three steps, $\underline{(\lambda x.xx)\lambda yz.yz} \to_\beta \underline{(\lambda yz.yz)\lambda yz.yz} \to_\beta \lambda z.\underline{(\lambda yz.yz)z} \to_\beta \lambda zw.zw$ where in the last step the substitution incurs usage of the 5th (renaming) clause on slide 12, renaming the inner z into w to avoid a variable-capture (of 'the other z'). For clarity, we have underlined in each step 'where' the $\to_\beta$-step takes place.*

Remark 3 *In λ-calculus a natural number n can be represented as the term $\lambda yz.y^n z$. Then 1 is represented by $\lambda yz.yz$ as in the exercises. Applying two such so-called Church-numerals to each other corresponds to exponentiation. From that perspective, the exercise can be seen to ask for computing 1 to the power 1, which indeed results in 1.*

Exercise 4 *We provide two answers, one based on normal forms (*syntactic*), the other (*semantic*) on Kripke models. where as in the exercise we interpret τ being arbitrary as knowing nothing about the structure of τ, and model that by assuming τ is some base type.*

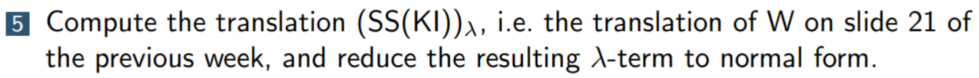
For the syntactic normal-form-based proof, we show that $(\tau \to \tau) \to \tau$ does not have closed inhabitants by reasoning as in Exercise 2:

- *a closed term M of that type must have shape $\lambda x.N$ with x assigned type $\tau \to \tau$ and N type τ (it cannot have shape $z\vec{Q}$ as then z would be free; and $(\lambda z.Q)\vec{Q}$ would not be in normal form);*
- *By the assumption that τ is a base type, N must be of shape $z\vec{P}$ (it cannot be an abstraction by τ being a base type, and as in the previous item $(\lambda z.Q)\vec{Q}$ would not be in normal form), with z not a free variable. This leaves $z = x$ as only possibility. Since x was assigned type $\tau \to \tau$ and N type τ, we conclude $\vec{P}$ is a vector of λ-terms of length 1, say P, to which the type τ is assigned.*
- *The reasoning for P is exactly the same as for N in the previous item. That is, we end up in a loop/an infinite regress and never obtain an actual term.*

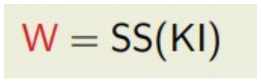
We conclude that there is no λ-term M in normal form of the specified type, thus by normalization of the simply typed λ-calculus, there is überhaupt no such λ-term M.

For the semantic Kripke-model-based proof, we give a countermodel for the propositional formula $(\tau \supset \tau) \supset \tau$ for τ a propositional letter (corresponding to *that τ is of base type). From this we conclude, by soundness of ND with respect to Kripke semantics, that that formula is not provable in ND (for intuitionistic logic), and hence by the Curry–Howard correspondence that the corresponding type is not inhabited. As countermodel, we take a 1-point model with world c where nothing is forced. To see that c does not force $(\tau \supset \tau) \supset \tau$, note that the assumption $\tau \supset \tau$ is always trivially forced (since any world that forces τ, forces τ), so in particular is forced in c, but c does not force τ per construction of this Kripke model.*

Exercise 5



5 Compute the translation $(\mathsf{SS(KI)})_\lambda$, i.e. the translation of W on slide 21 of the previous week, and reduce the resulting λ-term to normal form.



$\mathsf{W} = \mathsf{SS(KI)}$

I := λx.x
K := λx.λy.x
S := λx.λy.λz.x z (y z)

[(λx.λy.λz.(x z) (y z))(λx.λy.λz.(x z) (y z))][(λx.λy.x)(λx.x)]
→2 [(λy.λz.((λx0.λy0.λz0.(x0 z0) (y0 z0)) z) (y z))][λyx.x]
→ [(λy.λz.(λy0.λz0.(z z0) (y0 z0)) (y z))][λyx.x]
→ [λy.λz.(λz0.(z z0) ((y z) z0))][λyx.x]
→ [λy.λz.(λz0.(z z0) ((y z) z0))][λy1x1.x1]
→ [λz.(λz0.(z z0) (((λy1x1.x1) z) z0))]
→ [λz.(λz0.(z z0) (((λx1.x1)) z0))]
→ [λz.(λz0.(z z0) z0)]

Remark 4 *Using slide 21 we already know that the normal form must be $\lambda xy.x\,y\,y$, i.e. the normal form given there, $x\,y\,y$, preceded by λ-abstractions for x and y.*

Beware to put parentheses appropriately in translations. E.g., in the translation $(K\,I)_\lambda$ it is safest to first put parentheses around all translated elements, $((K)\,(I))_\lambda$ to yield $(\lambda xy.x)\,(\lambda x.x)$. Note that translating it as $\lambda xy.x\,\lambda x.x$ would be wrong, as that would, per convention, be parse as the entirely different λ-term $(\lambda x.(\lambda y.(x\,(\lambda x.x))))$.

This is not different from what we are used to in mathematics and computer science. For instance, if in mathematics we write $x{\cdot}y$ where $x = 1{+}2$ and $y = 3{+}4$, we mean the expression obtained by first putting parentheses, like $(x) + (y)$ and then substituting to yield $(1 + 2) \cdot (3 + 4)$, not the entirely different expression $1 + 2 \cdot 3 + 4$ which would then, per convention, be parsed as $((1 + (2 \cdot 3)) + 4)$. Also, when expanding code by replacing function calls we proceed like this. E.g. expanding `double(x)` *into its function body* `x+x` *in the expression* `double(x)^2` *we make sure to make* `x+x` *into an expression/code block first.*

Exercise 6

Part 1:

The problem of the assignment rules is that there is no way to rename variables. Due to this and the way λ-abstraction is defined, the same variable x is used for both λs. A way to circumvent this would be to allow for variable renaming similar to the α-step.

Part 2:

The type of `(\x -> \x -> x)` is Haskell is `p1 -> p2 -> p2`, while `(\x x -> x)` throws an error, so it isn't the same. This is because it is not allowed to use the same variable multiple times in the same λ-abstraction. The reason for this is obvious, as even if you would rename the variables, it would be unclear which x in the λ-abstraction belongs to the first and which to the second variable.

Remark 5

1. *Simply allowing variables to have different types assigned to them in contexts would* not *directly be a solution as that would have adverse effects. E.g. it would allow inferences such as:*

$$\dfrac{\dfrac{\dfrac{x:\tau, x:\tau\to\sigma \vdash x:\tau\to\sigma \qquad x:\tau, x:\tau\to\sigma\vdash x:\tau}{x:\tau,x:\tau\to\sigma\vdash x\,x:\sigma}}{x:\tau\vdash \lambda x.x\,x:(\tau\to\sigma)\to\tau}}{\vdash \lambda xx.x\,x:\tau\to(\tau\to\sigma)\to\tau}$$

   where it is unclear how the xs in the body $x\,x$ are linked to/bound by the xs in the respective λ-abstractions.

2. *Section 3 of Chapter 3 of the Haskell report specifies that for multiple λ-abstractions "the set of patterns (here: abstracted variables) must be* linear *— no variable may appear more than once in the set".*[3] *Unfortunately, it is not specified why this is enforced. It is only per convention that in* `\x -> (\x -> x)` *the* `x` *in the body is linked to (bound by) the second* `\x` *(and already allowing such λ-expressions causes the above problems). I see no reason why we could not have a similar convention for* `\x x -> x`*, corresponding to viewing contexts as lists/stacks instead of as sets, and to using stacks for parameter-passing when implementing function/procedure-calls in compilers.*

Exercise 7 *The problem and its resolution are described in the following solution:*

[3] From a mathematical point of view this specification makes no sense, since elements do not have a multiplicity in *sets*. An element either occurs or it doesn't; it doesn't occur twice or thrice or ... in a set.

```
to be a proper inductive defintion we need a base case
note that \vec{N} can be empty and in this case we have:
M is SC if M is SN (just remove all the \vec{N} stuff from the
                    given definition as it is empty)
this no longer recursively references SC. note however that the property
must hold for all \vec{N}, not just one cherry picked one
thus we need to ensure that being empty is the only possible case for \vec{N}
for the base case. for this we turn to types
if the type of M is a literal (no applicaion possible), then \vec{N} must be
empty and the above holds for all \vec{N}. this completes the induction
base which is already a special case of the induction step given in
the slides (type of M is a literal)
```

Remark 6 *Note that (higher-order) variables are* not *the base case (but terms of base type are). For instanc, if o is a base type, and $x : o \to o$, then SC of x is defined as that x is SN and xN is SN for all SC terms $N : o$.*

Defining SC per type *it is ok, since we only rely on SC of* smaller *types. In the base case, there are no smaller types, so the vector is empty, and the definition of SC relies on nothing. This allows the definition to get off the ground.*

Exercise 8 *That SC entails SN is trivial, since M being SC is defined to mean $M\vec{N}$ for all vectors $\vec{N}$ of SC terms. In particular, this applies to the empty vector, but M applied to the empty vector is just M itself.*

To see that every typed variable x is SC we must show that for every vector $\vec{N}$ of SC terms, $x\vec{N}$ is SN again. By the above, the assumption entails that $\vec{N}$ is a vector of SN terms. Now, since $\to_\beta$ only changes subterms of shape $(\lambda x.P)Q$, and such subterms can only occur in individual N_is, we obtain that if $x\vec{N} \to_w M$ then $M = x\vec{N}'$ for some vector $\vec{N}'$ of terms, of the same length as $\vec{N}$ and such that for each i, either $N_i = N_i'$ or $N_i \to_w N_i'$. Therefore, if there were an infinite reduction from $x\vec{N}$ there would be, by the Pigeon Hole Principle, an i such that the second of the two case holds. But that would give rise to an infinite $\to_w$ reduction from N_i, contradicting that N_i is SN.

Exercise 9 *Two things need to be supplemented: the (terminating) algorithm for actually finding inhabitants, and the cases of the subformula property.*

We start with the latter. The remaining, non-interesting, cases are:

- *The variable case, $\Gamma \vdash x : \phi$. The statement is trivially tautological (the formulas occurring are the formulas occurring).*
- *The abstraction case, $\Gamma \vdash (\lambda x.M) : \phi \to \psi$, is inferred from $\Gamma, x : \phi \vdash M : \psi$. By the IH all formulas χ occurring in the inference of the latter are subformulas of Γ, ϕ, ψ. Then all those χ are subformulas of $\Gamma, \phi \to \psi$ (since ϕ, ψ are subformulas of $\phi \to \psi$ and subformulas of subformulas are subformulas). Moreover, $\phi \to \psi$ is a subformula of itself.*

To give a terminating algorithm, we note that the subformula property guarantees that viewing the set[4] *of formulas in the context paired with the derived formula as a* state*, a brute-force bottom-up proof search algorithm works: We simply try all possible inference rules, not exploring further if a formula would be obtained that is not a subformula of the original state, or if a state is obtained that has been obtained before (on the same path; loop-checking). The algorithm terminates since there are only finitely many states.*

Exercise 11 `is still strongly normalizing as J has to be SC`

```
J M_1 M_2 ... M_n -> M_j M_k ... M_l
as CL is SC M_i are all SC thus the right side is SN by
definition of SC (and reachable in 1 step)
```

[4]We take sets since, e.g., lists would hinder loop-checking, as one might infinitely often introduce 'the same' formula.