Last Name: First Name:

Matriculation Number:

Retake Exam
KRMT

October 2024

	To be used for grading, do not write here										
prob.	1.1	1.2	1.3	2.1	2.2	3.1	3.2	3.3	3.4	Sum	grade
total	11	9	9	10	11	11	10	10	9	90	
reached											

1 MMT and LF

Problem 1.1 (Type System)

Consider the following LF theory:

```
a: type
b: a ⟶ type
c: {x:a} b x ⟶ type

r: a
s: {x:a} b x
t: {x:a} {y: b x} c x y
```

Relative to that theory:

1. Using an example from above, briefly explain (in at most two sentences) the concept of dependent types. 2 pt

   Solution: Dependent types are type expressions that have terms as subexpressions. An example is the type `b r` that contains the term `r`.

2. Check all typing judgments that hold: 3 pt
   - ☐ `[x:a][y: b x]x : a ⟶ b x ⟶ a`
   - ☑ `[x: a ⟶ a]t (x r) (s (x r)) : {x:a ⟶ a}c (x r) (s (x r))`
   - ☑ `t r (s r) : c r (s r)`
3. Give the type of the following term: `[x][y]t x y` 3 pt

   Solution: `{x:a}{y: b x}c x y`

4. Give a type that has exactly 2 distinct terms. 3 pt

   Solution: The type `a ⟶ a` only has 2 terms: `[x:a]r` and `[x:a]x`. (This is because `r` is the only way to create a term of type `a`, except for using the declared variables.)

Problem 1.2 (Notations and Type Inference)

Consider the following LF theory about matrix addition:

```
nat: type
zero: nat # 0 prec 0
succ: nat ⟶ nat # 1 ’ prec 50

matrix: nat ⟶ nat ⟶ type # 1 @ 2 prec 0
plus: {m,n} m@n ⟶ m@n ⟶ m@n # 3 + 4 prec 10
ex: 0’@0’
```

Relative to that theory:

1. Briefly (in at most 3 sentences) explain the declaration of `plus` regarding dependent typing, implicit arguments, and notations. 2 pt

---

Solution: The constant takes 4 arguments, first `m` and `n` and then two arguments whose type (as well as the return type) `m@n` depends on the previous two arguments. The notation makes the first two arguments implicit (by not mentioning argument positions `1` and `2`), i.e., they must be inferred from the type of the other two arguments.

---

2. Give the internal representation of the following terms. (The internal representation is the one where the notations are not used at all and all variable types are given.) 2 pt
   1. `[m,n,a: m@n,b]a+b`

      Solution: `[m:nat,n:nat,a: matrix m n, b: matrix m n]plus m n a b`

   2. `ex+ex`

      Solution: `plus (succ zero) (succ zero) ex ex`

3. Consider the string `[m,n,a]ex+a`. Explain (in at most two sentences) the result of applying type inference to it. 2 pt

---

Solution: The type of `a` is inferred to be `0'@0'`. But the types of `m` and `n` cannot be inferred, resulting in an error.

---

4. Give a declaration with type and notation for a constant `times` representing matrix multiplication. It should be written with an infix operator `*` and bind more tightly than `plus`. In the type, use the notations and omit all inferable information. 3 pt

---

Solution: `times: {l,m,n}l@m ⟶ m@n ⟶ l@n #4 * 5 prec 20`

Problem 1.3 (Theory Morphisms)

Consider the following LF theories and views

```
theory A =
  a: type
  b: type
  e: a
  f: a ⟶ b

theory B =
  nat: type
  z: nat
  s: nat ⟶ nat
```

```
view M : A → B =
  a = nat
  b = nat
  e = z
  f = [x] s x

view N : A → B =
  a = nat ⟶ nat
  b = nat
  e = [x] x
  f = [x] x z

view O : B → A =
  nat = (a ⟶ a) ⟶ (a ⟶ a)

  z =

  s = [p,q,r] (p q) r
```

1. Briefly explain (in about three sentences) the principle and relevance of type preservation along theory morphisms. 3 pt

Solution: A morphism `K` from `S` to `T` guarantees that if `t:A` holds over `S`, then `K(t):K(A)` holds over `T`. That can be used to move results from `S` to `T`. That enables modularization with enables using small theories that are reused as needed.

2. Give the fully β-reduced result of applying the morphism `N` to the term `f e`. 2 pt

Solution: `z`

3. Give the expected type for `s` in the morphism `O`. 2 pt

Solution: `((a ⟶ a) ⟶ (a ⟶ a)) ⟶ ((a ⟶ a) ⟶ (a ⟶ a))`

4. Give any term that can be assigned to `z` in the morphism `O`. 2 pt

Solution: e.g., `[q,r]r` or `[q,r]e` or `[q]q`

2 Logics

Problem 2.1 (General Concepts)

Consider the following LF theory

```
prop : type
false: prop
proof: prop ⟶ type
```

1. Using the above as an example, briefly explain (in at most three sentences) the idea of the proofs-as-terms representation in LF. 2 pt

Solution: Given a proposition `F:prop`, the type `proof F` holds the proofs of `F`. In particular, checking of proofs is represented in terms of type-checking in the logical framework, and `F` is a theorem if that type is non-empty.

2. Briefly explain (in at most two sentences) the key relation between the proofs-as-terms representation and the type preservation along morphisms. 2 pt

Solution: Because morphisms preserve typing, proofs `P:proof F` are mapped to proofs `M(P): proof M(F)`. In particular, theorems are mapped to theorems.

3. Explain the proof rule corresponding to the type `proof false ⟶ {F}proof F`. 2 pt

Solution: This is the false-elimination rule. It captures that if `false` is provable, so is every other formula.

4. Give the formalizations of untyped and typed universal quantification. Include the declarations of all constants you need that are not yet given above. 4 pt

Solution: The untyped representation is relative to a fixed type `term: type` and uses `forall: (term ⟶ prop) ⟶ prop`. The type representation is relative a set of type `tp: type` and terms `tm: tp ⟶ type` of each type, and it uses `{A}(tm A ⟶ prop) ⟶ prop`.

Problem 2.2 (Connectives and Proof Rules)

Consider the following partial formalization of propositional logic:

```
prop : type
proof: prop ⟶ type # proof 1 prec -5
not : prop ⟶ prop # ¬ 1
conj: prop ⟶ prop ⟶ prop # 1 ∧ 2
disj: prop ⟶ prop ⟶ prop # 1 ∨ 2
```

1. Give the introduction and elimination rules for conjunction. 3 pt

Solution:

```
conjI: {a,b} proof a ⟶ proof b ⟶ proof (a ∧ b)
conjEl: {a,b} proof (a ∧ b) ⟶ proof a
conjEr: {a,b} proof (a ∧ b) ⟶ proof b
```

---

2. Briefly explain (in at most two sentences) the analogy between a proof by contradiction and a formal proof using negation introduction. 2 pt

---

Solution: A proof by contradiction establishes a conjecture by assuming it is false and deriving a contradiction. Negation introduction captures that principle for the special case where the conjecture is a negated formula.

---

3. We want to extend the above theory with a **ternary** disjunction operator D such that $D\ a\ b\ c$ is true if at least one of its three arguments is true. Give a definition (including type and definiens) for it. 2 pt

---

Solution: `D: prop ⟶ prop ⟶ prop ⟶ prop = [a,b,c]a∨ b ∨ c`

---

4. Continuing the previous question, give the introduction and elimination rules that characterize ternary disjunction and that can be defined from the rules for binary disjunction. Give only the types, do not include the definitions. 4 pt

---

Solution:

```
DIl: {a,b,c} proof a ⟶ proof D a b c
DIm: {a,b,c} proof b ⟶ proof D a b c
DIr: {a,b,c} proof c ⟶ proof D a b c
DE: {a,b,c,g} (proof a ⟶ proof g) ⟶ (proof b ⟶ proof g) ⟶ (proof c ⟶ proof g)
  ⟶ (proof D a b c ⟶ proof g)
```

---

3 Mathematical Domains

Problem 3.1 (Monoids)

Consider the following theories:

```
theory CommMonoid : FOL =
  op: term ⟶ term ⟶ term # 1 * 2 prec 100
  e: term
  assoc: proof ∀ [x] ∀ [y] ∀ [z] (x*y)*z ≐ x*(y*z)
  neut: proof ∀ [x] x*e ≐ x

  comm: ???

theory LeftBoundedRelation : FOL =
  op: term ⟶ term ⟶ prop
  bound: term
  bounded: proof ∀ [x] op bound x
```

1. Give the missing type of the constant `comm` to obtain a theory for commutative monoids. 2 pt

Solution: `comm: proof ∀ [x] ∀ [y]x*y ≐ y*x`

2. Assume, alternatively, we wanted to give the theory of (not necessarily commutative) monoids. How would we have to change the above formalization? 2 pt

Solution: Remove the axiom `comm` and add the other neutrality axiom (because it is not redundant anymore in the absence of commutativity).

3. Give a theory for commutative groups that includes `CommMonoid`. 3 pt

Solution:

```
theory CommGroup =
  include CommMonoid
  inv: term ⟶ term
  invAx: proof ∀ [x] x*(inv x) ≐ e
```

4. Give a morphism `Divides` that shows that every commutative monoid allows defining the left-bounded relation that holds for (x, y) if there is an f such that $x * f = y$. For the axioms, only give the expected type, not the proofs. 4 pt

Solution:

```
view Divides: LeftBoundeRelation → CommMonoid =
  op = [x,y] ∃ [f] x*f ≐ y
  bound = e
  bounded: proof ∀ [x] ∃ [f] e*f ≐ f
```

Problem 3.2 (Lattices)

Lattices are algebraic objects with two binary operations, called join and meet, each of which must be a semilattice. The following is a partial formalization (with some types omitted):

```
theory Semilattice : FOL =
  op: term ⟶ term ⟶ term
  idempotent : proof ∀ [x] op x x  ≐ x
  associative: ...
  commutative: ...

theory Lattice : FOL =
  structure join : Semilattice
  structure meet : Semilattice
  absorb_jm: ...
  absorb_mj: ...
```

1. Give the names and types of all constant declarations that are present in `Lattice` after elaborating the structures (not counting the ones provided by `FOL`). 4 pt
If types are omitted in the formalization, also omit them in your answer.

Solution:

```
join/op: term ⟶ term ⟶ term
join/idempotent: proof ∀ [x] join/op x x ≐ x
join/associative: ...
join/commutative: ...
meet/op: term ⟶ term ⟶ term
meet/idempotent: proof ∀ [x] meet/op x x ≐ x
meet/associative: ...
meet/commutative: ...
```

---

2. The axiom `absorb_jm` is meant to formalize the property $x \sqcup (x \sqcap y) = x$ where $\sqcup$ and $\sqcap$ are the join and meet operations of the lattice, respectively. Add its type to the above formalization. 2 pt

---

Solution:

```
absorb_jm: proof ∀ [x] ∀ [y] join/op x (meet/op x y) ≐ x
```

---

3. Assume we have a `theory Nat : FOL` in which the terms represent natural numbers and in which the usual operations on natural numbers are declared including binary operations `min` and `max` for the minimum and maximum of two numbers. 4 pt
These form a lattice if minimum and maximum are interpreted as join and meet, respectively.
Give the views that yield a modular view `NatLat : Lattice → Nat` representing that interpretation. Omit all assignments to axioms.

Solution:

```
view NatMeet : Semilattice → Nat =
  op = min
  ...

view NatJoin : Semilattice → Nat =
  op = max
  ...

view NatLat : Lattice → Nat =
  structure meet = NatMeet
  structure join = NatJoin
  ...
```

Problem 3.3 (Numbers)

Consider the following theory for natural numbers with infinity:

```
theory NatInf : FOL =
  nat = term
  z: nat
  s: nat ⟶ nat
  i: nat

  si : ⊢ s i  ≐ i
```

1. Assuming equality axioms are left-to-right rewrite rules, what are the canonical forms here? 3 pt

   Solution: `i`, `z`, `s z`, `s s z`, …

2. Extend the formalization with addition: declare a binary operation for addition and axioms that determine the result of addition for all arguments. 4 pt

   Solution: e.g.,

   ```
     plus: nat ⟶ nat ⟶ nat # 1 + 2
     plus_z : {x} ⊢ z +x  ≐ x
     plus_s : {x,y} ⊢ (s y)+x  ≐ s(y+x)
     plus_i : {x} ⊢ i +x  ≐ i
   ```

3. Give the formalization of an induction schema that can be used to prove a property for all elements of the set of natural numbers with infinity. 3 pt

   Solution: e.g.,

   ```
   induction: {P: term ⟶ prop}
    ⊢ P z
    ⟶ ⊢ P i
    ⟶ ({x} ⊢ P x ⟶ ⊢ P (s x))
    ⟶ {x} ⊢ P x
   ```

Problem 3.4 (Set Theory)

Consider the following fragment of a formalization of set theory

```
theory SetTheory : FOL =
  set = term
  in : set ⟶ set ⟶ prop # 1 ∈ 2

  axiom: proof  ∀ [x]  ∀ exists [p]  ∀ [s] (forall [u] u ∈ s ⇒ u ∈ x) ⇒ s ∈ p
```

1. Explain the intuition behind the stated axiom. 2 pt

Solution: It expresses that the powerset of x exists for all x.

2. Add a definition of the binary predicate on sets s that expresses that the two sets are disjoint. 2 pt

Solution: `disjoint: term ⟶ term ⟶ prop = [x,y]¬ ∃ [z] z ∈ x ∧ z ∈ y`
A definition that assumes the empty set has already been defined was also accepted.

3. Briefly explain (in at most two sentences) the pros and cons of assuming a description operator when formalizing set theory in first-order logic. 2 pt

Solution: pro: It is possible to formalize set theory without any primitive function symbols and define all operations later based on existential axioms.
con: The logic is more complex than standard first-order logic.

4. Assume we have already defined the following operations 3 pt

```
cartesian_product : set ⟶ set ⟶ set # 1 * 2
bigunion : set ⟶ set
biginter : set ⟶ set
powerset : set ⟶ set
ordered_pair : set ⟶ set ⟶ set # 1 , 2
separation : set ⟶ (set ⟶ prop) ⟶ set # 1 | 2
replacement : set ⟶ (set ⟶ set) ⟶ set # 1 repl 2
existsUnique : (set ⟶ prop) ⟶ prop
```

Define the operator B^A that returns the set of functions from A to B (giving type, definition, and a reasonable notation).

Solution: e.g.,

```
fun : set ⟶ set ⟶ set
    = [A,B] (powerset A*B) | [r] ∀ [a] a ∈ A ⇒ existsUnique [b] (a,b) ∈ r
    # 2 ^ 1
```