Last Name: First Name:

Matriculation Number:

Final Exam KRMT – SS 2023

Sep 25, 2023

Please ignore the QR codes; do not write on them, they are for grading support

	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	10	10	10	10	10	10	10	10	10	90	
reached											

The “solutions” to the exam/assignment problems in this document are supplied to give students a starting point for answering questions. While we are striving for helpful “solutions”, they can be incomplete and can even contain errors even after our best efforts.

In any case, grading student’s answers is not a process of simply “comparing with the reference solution”, therefore errors in the “solutions” are not a problem in this case.

If you find “solutions” you do not understand or you find incorrect, discuss this on the course forum and/or with your TA and/notify the instructors. We will – if needed – correct them ASAP.

1 MMT and LF

Problem 1.1 (Type System)

Consider the following LF theory:

```
a: type
b: a ⟶ type

e: a
f: a ⟶ a

g: {x:a} b x ⟶ b x
h: {x:a} (b x ⟶ b x) ⟶ b (f x)
```

Relative to that theory:

1. Taking an example from the above, briefly explain (in at most three sentences) the concept of a higher-order function. 2 Points

   Solution: A higher-order function takes other functions as arguments, e.g., the second argument of `h` above.

2. Check all typing judgments that hold: 3 Points
   - ☐ `f (f (f e)) : a`[1]
   - ☐ `g e (g e (f e)) : b e`[2]
   - ☐ `g (f e) (b e) : b (f e)`[3]

3. Give the type of the following term: `h e ([x:b e]g e x)` 3 Points

   Solution: `b (f e)`

4. Give an empty type. 2 Points

   Solution: e.g., `b e`

Problem 1.2 (Notations and Type Inference)

Consider the following LF theory:

```
nat: type
tp : type
vector: tp ⟶ nat ⟶ tp # 1 ^ 2 prec 50
```

[1]Correct
[2]Wrong
[3]Wrong

```
p: nat ⟶ nat ⟶ nat # 1 + 2 prec 10
t: nat ⟶ nat ⟶ nat # 1 * 2 prec 20

vp: {a,n} a^n ⟶ a^n ⟶ a^n # 3 ⊕ 4 prec 5
```

Relative to that theory:

1. Briefly explain (in at most three sentences) why the notation for `vp` does not mention argument positions 1 and 2 and how that affects the parsing/type-checking process. 2 Points

   Solution: Their values can be inferred from the other arguments. The system inserts a placeholder for the unknown arguments and determines their values during type-checking.

2. Give the internal representation of the following terms. (The internal representation is the one where the notations are not used at all, e.g., the internal representation of `x+y` is `p x y`.) 3 Points

   1. `x*y`

      Solution: `t x y`

   2. `x+y*z`

      Solution: `p x (t y z)`

   3. `v ⊕ w` in a context where `v` and `w` have type `a^n`

      Solution: `vp a n v w`

3. Consider the term `[w,x,y, z: w^(x+y)]z ⊕ z`. 2 Points

   1. Give the type of the variable `w`.

      Solution: `tp`

   2. Give the type of the subterm `z ⊕ z`.

Solution: `w^(x+y)` or `vector w (p x y)`

4. Give a declaration for a constant `vc` that takes two vectors `v`,`w` over the same type and of possibly different dimensions and returns their concatenation written `v@w`. It should bind as tightly as `vp`. 3 Points

Solution: `vc: {a,m,n}a^m ⟶ a^n ⟶ a^(m+n) #4 @ 5 prec 5`

Problem 1.3 (Theory Morphisms)

Consider the following LF theories and views

```
theory A =
  n: type
  z: n
  s: n ⟶ n

theory B =
  t: type
  a: t ⟶ t
```

```
view M : A → B =
  n = t ⟶ t
  z = [x] x
  s = [f] [x] a (f x)

view N : B → A =
  t = n
  a = s

view O : B → A =
  t = n ⟶ n
  a = (omitted)
```

1. Briefly explain (in at most three sentences) the purposes of grouping declarations into theories. 2 Points

Solution: Every theory encapsulates a separate context so that expressions in different contexts can be formalized in parallel. Moreover, theories can be reused as a whole, e.g., when building theories modularly.

2. Give the result of applying the morphism `N M N` to the type `t`. 2 Points

Solution: `n ⟶ n`

3. Give the expected type for `s` in the morphism `M`. 2 Points

Solution: `(t ⟶ t) ⟶ (t ⟶ t)`

4. Consider a term `u` of type `U` over theory `A`. What does the type preservation property of the theory morphism `M` imply in this case? 2 Points

---

Solution: `M(u):M(U)` over theory `B`

---

5. Give any term that completes the morphism `O`. 2 Points

Solution: e.g., `[f:n ⟶ n]f` or `[f:n ⟶ n]s`

---

2 Logics

Problem 2.1 (General Concepts)

Consider the following LF theory

```
prop : type
proof: prop ⟶ type
term : type
```

1. Briefly explain (in at most three sentences) the idea behind the proofs-as-terms representation. 2 Points

---

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 three sentences) the purpose of using a logical framework. 2 Points

---

Solution: A logical framework allows formally defining and reasoning about the syntax and semantics of logics and related systems.

---

3. What is the importance of the type preservation property of theory morphisms when representing proofs? Answer in one sentence. 2 Points

---

Solution: Theory morphisms map proofs to proofs and thus theorems to theorems.

---

4. Give a type that can be used to represent inconsistency. Briefly explain (in at most two sentences) why it can be used. 2 Points

   Solution: The type `{F:prop}proof F` is inhabited iff there is a proof for every proposition.

5. Give the types of 2 Points

   1. universal quantification

      Solution: e.g. `(term ⟶ prop) ⟶ prop`

   2. equality

      Solution: e.g. `term ⟶ term ⟶ 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. Briefly explain (in at most two sentences) the difference between an introduction and an elimination rule. 2 Points

   Solution: An introduction rule expresses how to prove a proposition. An elimination rule expresses how to use it to prove others.

2. Give the introduction rule for conjunction. 2 Points

   Solution: `conjI: {a,b}proof a ⟶ proof b ⟶ proof a ∧ b`

3. We want to extend the above theory with a primitive binary connective `x L y` that is true if and only if its first argument `x` is true. Give a declaration for it, including type and notation (but no definiens). Also give appropriate introduction and elimination proof rules for it. 4 Points

Solution: e.g.,

```
left: prop ⟶ prop ⟶ prop # 1 L 2
leftI: {F,G} proof F ⟶  proof F L G
leftE: {F,G} proof (F L G) ⟶ proof F
```

4. Give a definition (including definiens and notation) that defines the binary connective `x ≠ y` that is true iff both arguments have different truth values. 2 Points

Solution: `diff: prop ⟶ prop ⟶ prop = [x,y](x∧¬ y) ∨(¬ x∧ y) # 1 ≠ 2`

3 Mathematical Domains

Problem 3.1 (Monoids)

Consider the following theories

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

1. Give the axiom(s) that is/are missing for this to be an axiomatization of monoids. 2 Points

Solution: `neut: proof ∀ [x]x*e=x ∧ e*x=x`

2. Give a theory for groups that includes `Monoid`. 3 Points

Solution:

```
theory Group =
  include Monoid
  inv: univ ⟶ univ
  invax: proof ∀ [x] x*(inv x) = e
```

Either one of the inversion axioms is sufficient. But giving both is not wrong.

3. Give a morphism `Opp` from `Monoid` to itself that maps `x*y` to `y*x`. You do not have to do any proofs — instead, just give the expected type and omit the proof. 3 Points

Solution:

```
view Opp : Monoid → Monoid =
  op = [x,y]y*x
  e = e
  assoc : proof  ∀ [x]  ∀ [y]  ∀ [z] z*(y*x) = (z*y)*x
```

4. Draw the theory graph involving all of the above (except `FOL`). 2 Points

Solution: `Op` ↻ `Monoid` → `Group`

Problem 3.2 (Rings)

Consider the following theory

```
theory Magma : FOL =
  op: term ⟶ term ⟶ term

theory BiMagma : FOL =
  structure add : Magma
  structure mul : Magma
```

1. Briefly explain (in at most two sentences) why that theory uses `structure` instead of `include`. 2 Points

Solution: Two includes of the same theory are redundant and would identify the two magmas. Structures create two separate magmas.

2. Give the name and type of all constant declarations that are present in `BiMagma` (not counting the ones provided by `FOL`) after elaborating the structures. 2 Points

Solution:

```
add/op: term ⟶ term ⟶ term
mul/op: term ⟶ term ⟶ term
```

3. Give the proposition in `BiMagma` that expresses that multiplication on the left distributes over addition. 3 Points

---

Solution:

```
 ∀ [x]  ∀ [y]  ∀ [z]
 mul/op x (add/op y z) = add/op (mul/op x y) (mul/op x z)
```

---

4. Assume we have a `theory Int : FOL` in which the terms represent integer numbers and in which the usual operations on integers such as `0,1,+,-,*` are declared. Give the views that yield a modular view `IntBM : BiMagma → Int`. 3 Points

---

Solution:

```
view IntAdd : Magma → Int =
  op = +

view IntMul : Magma → Int =
  op = *

view IntBM : BiMagma → Int =
  structure add = IntAdd
  structure mul = IntMul
```

Problem 3.3 (Numbers)

Consider the following theory

```
theory Nat : FOL =
  nat = term
  z: nat
  s: nat ⟶ nat

  plus: nat ⟶ nat ⟶ nat # 1 + 2
  plus_z : {m} proof z+m = m
  plus_s : {m,n} proof (s m)+n = s(m+n)
```

1. Here addition is specified by equality axioms that allow reducing expressions to canonical forms. 2 Points
   1. What are the canonical forms for type `nat` here?

Solution: All terms built from only `z` and `s`.

---

2. What does “reducing to canonical forms” mean in this case? Answer in at most two sentences.

   Solution: Using the equality axioms left-to-right, any application of `plus` to canonical forms can be rewritten into a canonical form.

2. Briefly explain (in at most four sentences) the purpose of the no-junk and no-confusion axioms (which are omitted above) that capture the inductive structure of the natural numbers. In doing so, sketch what the axioms look like for the natural numbers. 4 Points

   Solution: The no-confusion axioms state that all canonical forms are different, i.e., that `z` is not equal to any `s x` and that `s` is injective. The no-junk axiom captures that there are no other natural numbers than the canonical forms: any property that holds for all canonical forms holds for all natural numbers.

3. Add a declaration and equality axioms that formalize the binary operation of absolute difference (e.g., the function $|m - n|$). 4 Points

   Solution: e.g.,

```
ad: nat ⟶ nat ⟶ nat # 1 - 2
ad_any_z: {m} proof m-z = m
ad_z_any: {m} proof z-m = m
ad_s_s: {m,n} proof (s m)-(s n) = m-n
```

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

  extensionality: proof ∀ [x] ∀ [y]
    (forall [z] z ∈ x ⇔ z ∈ y) ⇒ x=y
```

1. Explain the axiom of extensionality. 2 Points

   Solution: It states that two sets are equal if they contain the same elements.

2. Give the formalization of the axiom that the empty set exists. 2 Points

Solution: `emptyExists: proof ∃ [e] ¬∃ [x] x ∈ e`

3. Briefly explain (in at most three sentences) the difficulty of using standard first-order logic for set theory when it comes to defining operations such as the empty set, and how adding a description operator can overcome this. 2 Points

Solution: Set theory can prove the existence of objects, but FOL cannot directly refer to them. The description operator is a logical feature that returns a uniquely existing object with a given property, thus allowing to refer to it.

4. Assume we have already defined the following operations 4 Points

```
empty: set
unordered_pair : set ⟶ set ⟶ set # 1 uop 2
bigunion: set ⟶ set
biginter: set ⟶ set
powerset: set ⟶ set
separation : set ⟶ (set ⟶ prop) ⟶ set # 1 | 2
replacement: set ⟶ (set ⟶ set) ⟶ set # 1 repl 2
```

Moreover, assume we have already defined a set `F` that contains all sets built using only the above operations. Define the set of natural numbers.

Hint: It helps to give a few auxiliary definitions. Eventually, take the intersection of all subsets of `F` that contain the natural numbers.

Solution:

```
z = empty
s : set ⟶ set = [x] bigunion (x uop (x uop x))
hasnats : set ⟶ prop = [h] z ∈ h ∧ ∀ [n] n ∈ h ⇒ (s n) ∈ h
nat = biginter (F | hasnats)
```