CIS 500 — Software Foundations

Midterm I

Answer key

October 11, 2006

Instructions

- This is a closed-book exam: you may not use any books or notes.
- You have 80 minutes to answer all of the questions. The entire exam is worth 80 points for students in section 002 and 90 points for students in section 001 (there is one PhD-section-only problem).
- Questions vary significantly in difficulty, and the point value of a given question is not always exactly proportional to its difficulty. Do not spend too much time on any one question.
- Partial credit will be given. All correct answers are short. The back side of each page may be used as a scratch pad.
- Good luck!

OCaml

1. (5 points) The `forall` function takes a predicate `p` (a one-argument function returning a boolean) and a list `l`; it returns `true` if `p` returns `true` on every element of `l` and `false` otherwise.

```
# forall (fun x -> x >= 3) [2;11;4];;
- : bool = false

# forall (fun x -> x >= 3) [3;4;5];;
- : bool = true
```

   (a) What is the type of `forall`? *Answer:* `('a → bool) → 'a list → bool`

   (b) Complete the following definition of `forall` as a recursive function: *Answer:*

```
let rec forall p l =
  match l with
    | [] -> true
    | h::t -> (p h) && forall p t
```

Grading scheme: (a) gets 0/2 for completely wrong type, 1/2 for small mistakes (e.g. not polymorphic type), 2/2 otherwise. (b) gets 0/3 for completely wrong functions, 1/3 if there is some pattern matching involved, 2/3 for simple mistakes such as base case error and 3/3 for various completely correct solutions.

2. (5 points) Recall the function `fold` discussed in class:

```
# let rec fold f l acc =
  match l with
    [] -> acc
  | a::l -> f a (fold f l acc);;
val fold : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Complete the following definition of `forall` by supplying appropriate arguments to `fold`:

Answer:

```
let forall p l = fold (fun x acc → (p x) && acc) l true
```

Grading scheme: Roughly one point for right number/type of arguments, one point for the right initial accumulator, three points for the function:

0: Blank.

1: Major omissions and errors: e.g., omitting f entirely, or wrong f and an integer as initial accumulator.

2: Correct number and types of arguments, but f wrong. Most common:
`let forall p l = fold p l true`

3: f argument has right type, or right body, but two minor errors.

4: One minor error.

5: Perfect.

Untyped lambda-calculus

The following questions are about the untyped lambda calculus. For reference, the definition of this language appears on page 13 at the end of the exam.

Recall the definitions of the following lambda-terms from the book and/or lecture notes:

```
/* A dummy "unit value", for forcing thunks */
unit = λx. x;

/* Standard definition of booleans */
tru = λt. λf. t;
fls = λt. λf. f;
not = λb. b fls tru;
test = λb. λt. λf. b t f unit;

/* Standard definition of pairs */
fst = λp. p tru;
snd = λp. p fls;
pair = λx. λy. λsel. sel x y;

/* Standard call-by-value fixed point function. */
fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y));

/* Standard definitions of church numerals and arithmetic operations */
c0 = λs. λz. z;
c1 = λs. λz. s z;
c2 = λs. λz. s (s z);
c3 = λs. λz. s (s (s z));
c4 = λs. λz. s (s (s (s z)));
c5 = λs. λz. s (s (s (s (s z))));
c6 = λs. λz. s (s (s (s (s (s z)))));
scc = λn. λs. λz. s (n s z);
iszro = λm. m (λdummy. fls) tru;
zz = pair c0 c0;
ss = λp. pair (snd p) (scc (snd p));
prd = λm. fst (m ss zz);
```

3. (6 points) Circle the term that each of the following lambda calculus terms steps to, using the *single-step* evaluation relation $\mathtt{t} \longrightarrow \mathtt{t}'$. If the term is a normal form, circle DOESN'T STEP.

(a) `(λx.x) (λx. x x) (λx. x x)`

i. `(λx. x) (λx. x x) (λx. x x)`
ii. `(λx. x x) (λx. x x)`
iii. `(λx′. (λx. x x)) (λx. x x)`
iv. `(λx. x) (λx. x x)`
v. DOESN'T STEP

Answer: (ii)

(b) `(λx. (λx.x) (λx. x x))`

i. `(λx. (λx.x) (λx. x x))`
ii. `(λx. (λx. x x))`
iii. `(λx. (λx. x))`
iv. `(λx. x) (λx. x x)`
v. DOESN'T STEP

Answer: (v)

(c) `(λx. (λz. λx. x z) x) (λx. x x)`

i. `(λx. (λz. λx. x z) x) (λx. x x)`
ii. `(λz. λx′. (λx. x x) z) (λx. x x)`
iii. `(λz. λx. x z) (λx. x x)`
iv. `(λx. x (λx. x x))`
v. DOESN'T STEP

Answer: (iii)

Grading scheme: Two points for each.

4. (10 points) Recall the definitions of observational and behavioral equivalence from the lecture notes:

   - Two terms `s` and `t` are *observationally equivalent* iff either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both are divergent.
   - Terms `s` and `t` are *behaviorally equivalent* iff, for every finite sequence of values $\texttt{v}_1$, $\texttt{v}_2$, ..., $\texttt{v}_n$ (including the empty sequence), the applications

     $$\texttt{s}\ \texttt{v}_1\ \texttt{v}_2\ \ldots\ \texttt{v}_n$$

     and

     $$\texttt{t}\ \texttt{v}_1\ \texttt{v}_2\ \ldots\ \texttt{v}_n$$

     are observationally equivalent.

   For each of the following pairs of terms, write *Yes* if the terms are behaviorally equivalent and *No* if they are not.

   (a) `plus` $\texttt{c}_2\ \texttt{c}_1$
   $\texttt{c}_3$
   Answer: Yes

   (b) `tru`
   `λx. λy. (λz. z) x`
   Answer: Yes

   (c) `λx. λy. x y`
   `λx. λy. x (λz. z) y`
   Answer: No

   (d) `(λx. x x) (λx. x x)`
   `λx. (λx. x x) (λx. x x)`
   Answer: No

   (e) `λx. λy. x y`
   `λx. x`
   Answer: Yes

   Grading scheme: Two points for each.

5. (12 points) Complete the following definition of a lambda-term `equal` that implements a *recursive* equality function on Church numerals. For example, `equal c0 c0` and `equal c2 c2` should be behaviorally equivalent to `tru`, while `equal c0 c1` and `equal c5 c0` should be behaviorally equivalent to `fls`. You may freely use the lambda-terms defined on page 3.

```
equal =
  fix (λe.
       λm. λn.
         test (iszro m)
ANSWER:
            (λdummy. (iszro n))
            (λdummy.
               test (not (iszro n))
                 (λdummy. e (prd m) (prd n))
                 (λdummy. fls)))
```

Grading scheme:

- *For getting the three main branches right:*
  - *2pts for first branch, need to test* `(iszro n)` *and return* `fls`
  - *in second branch, need to test* `(iszro n)` *and return (2pts)* `true` *or (2pts)* `(e (prd n) (prd m)`.
- *2pts for using* `test` *correctly (i.e., use of thunks)*
- *2pts for understanding how to right a recursive function with* `fix` *(i.e., use* `e`*, not* `equal`*)*
- *2pts for getting everything else right.*

Simple types for numbers and booleans

6. (18 points) Recall the following properties of the language of numbers and booleans:

   - **Progress**: If $\vdash$ `t : T`, then either `t` is a value or else `t` $\longrightarrow$ `t'` for some `t'`.
   - **Preservation**: If $\Gamma \vdash$ `t : T` and `t` $\longrightarrow$ `t'`, then $\Gamma \vdash$ `t' : T`.
   - **Uniqueness of types**: Each term `t` has at most one type, and if `t` has a type, then there is exactly one derivation of that typing.

   Each part of this exercise suggests a different way of changing the language of typed arithmetic and boolean expressions (see page 11 for reference). Note that these changes are not cumulative: each part starts from the original language. In each part, for each property, indicate (by circling TRUE or FALSE) whether the property remains true or becomes false after the suggested change. If a property becomes false, give a counterexample.

   (a) Suppose we add the following typing axiom:

   $$\texttt{pred (succ 0) : Bool}$$

   Progress: *Answer: True.*

   Preservation: *Answer: False.* `pred (succ 0)` *has type* `Bool`*, but reduces to* `0`*, which does not have type* `Bool`*.*

   Uniqueness of types: *Answer: False. Consider* `pred (succ 0)`*, which has types* `Nat` *and* `Bool`*.*

   (b) Suppose we add the following evaluation axiom:

   $$\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow t_1$$

   Progress: *Answer: True*

   Preservation: *Answer: False. Consider* `if true then 0 else 0`*, which has type* `Nat` *and steps to a term which has only type* `Bool`*.*

   Uniqueness of types: *Answer: True*

   (c) Suppose we add a new type `Foo` and two new typing rules:

   $$\frac{t_1 \texttt{ : Nat}}{\texttt{pred } t_1 \texttt{ : Foo}}$$

   $$\frac{t_1 \texttt{ : Foo}}{\texttt{succ } t_1 \texttt{ : Nat}}$$

   Progress: *Answer: True*

   Preservation: *Answer: False. Suppose we're given* `pred 0 : Foo`*. Then* `pred 0` $\longrightarrow$ `0`*, but* `0 : Nat`*.*

   Uniqueness of types: *Answer: False.* `pred 0 : Foo` *and* `pred 0: Nat` *are both derivable.*

   Grading scheme: For each property:

   - *-2 for* `True` *instead of* `False` *or vice versa*
   - *-1 for insufficient detail or not clearly explained counterexample*

7. (10 points) **[For students in the PhD section only.]** Suppose we add to the language of numbers and booleans two new types, called `True` and `False`, plus the following rules. (Note how the two rules for `if` allow types to be given to conditionals where the branches are not of the same type.)

$$\texttt{true} : \texttt{True}$$

$$\texttt{false} : \texttt{False}$$

$$\frac{t_1 : \texttt{True} \quad t_2 : T_2 \quad t_3 : T_3}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T_2}$$

$$\frac{t_1 : \texttt{False} \quad t_2 : T_2 \quad t_3 : T_3}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T_3}$$

Grading scheme:

- *Part (a) worth 4 points—2 for each type.*
- *Part (b) worth 6 points.*
  - *First case worth 2 points: binary.*
  - *Second case worth 4 points: no credit for a false statement, partial credit for a true but weaker statement or missing cases.*

(a) What type(s) can be derived for the following term?

```
if (if true then true else 0) then false else 0
```

Answer: `False` and `Bool`

(b) The *inversion lemma* tells us, for each syntactic form of terms, how terms of this form can be given types by the typing rules—intuitively, it allows us to "read the typing relation backwards."

Here is the inversion lemma from class for the original language of numbers and booleans:
Lemma:

- If `true` : `R`, then `R` = `Bool`.
- If `false` : `R`, then `R` = `Bool`.
- If `if` t_1 `then` t_2 `else` t_3 : `R`, then t_1 : `Bool`, t_2 : `R`, and t_3 : `R`.
- If `0` : `R`, then `R` = `Nat`.
- If `succ` t_1 : `R`, then `R` = `Nat` and t_1 : `Nat`.
- If `pred` t_1 : `R`, then `R` = `Nat` and t_1 : `Nat`.
- If `iszero` t_1 : `R`, then `R` = `Bool` and t_1 : `Nat`.

Complete the statements of the following clauses for the enriched language.

Lemma [Inversion]:

- If `true` : `T`, then *Answer: either* T = `Bool` *or* T = `True`.
- If `if` t_1 `then` t_2 `else` t_3 : `T`, then *Answer: either (1)* t_1 : `Bool` *and* t_2 : T *and* t_3 : T*, or (2)* t_1 : `True` *and* t_2 : T *and* t_3 : T_3 *for some* T_3*, or (3)* t_1 : `False` *and* t_2 : T_2 *for some* T_2 *and* t_3 : T.

Simply typed lambda-calculus

The following questions are about the simply typed lambda-calculus over the base type `Nat` (not `Bool`, as in the book!). For reference, the definition of this language appears on page 14 at the end of the exam.

8. (6 points) Draw a typing derivation for the statement

$$\emptyset \vdash (\lambda \texttt{f:Nat}\to\texttt{Nat. f 0})\ (\lambda \texttt{g:Nat. pred g}) : \texttt{Nat}$$

Answer:

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{(f:Nat}\to\texttt{Nat)} \in \texttt{(f:Nat}\to\texttt{Nat)}}{\texttt{f:Nat}\to\texttt{Nat} \vdash \texttt{f} : \texttt{Nat}\to\texttt{Nat}}\ \text{T-Var} \quad \dfrac{}{\texttt{f:Nat}\to\texttt{Nat} \vdash \texttt{0} : \texttt{Nat}}\ \text{T-Zero}}{\texttt{f:Nat}\to\texttt{Nat} \vdash \texttt{f 0} : \texttt{Nat}}\ \text{T-App}}{\emptyset \vdash (\lambda \texttt{f:Nat}\to\texttt{Nat. f 0}) : \texttt{(Nat}\to\texttt{Nat)}\to\texttt{Nat}}\ \text{T-Abs} \quad \dfrac{\dfrac{\dfrac{\texttt{(g:Nat)} \in \texttt{(g:Nat)}}{\texttt{g:Nat} \vdash \texttt{g} : \texttt{Nat}}\ \text{T-Var}}{\texttt{g:Nat} \vdash \texttt{pred g} : \texttt{Nat}}\ \text{T-Pred}}{\emptyset \vdash (\lambda \texttt{g:Nat. pred g}) : \texttt{Nat}\to\texttt{Nat}}\ \text{T-Abs}}{\emptyset \vdash (\lambda \texttt{f:Nat}\to\texttt{Nat. f 0})\ (\lambda \texttt{g:Nat. pred g}) : \texttt{Nat}}\ \text{T-App}$$

Grading scheme: Up to one point off for each missing rule, misapplied rule, or incorrect judgment in the internal part of the tree.

9. (18 points) Here are the weakening and permutation lemmas for $\lambda_{\rightarrow}$:

Lemma [Weakening]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{x} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \mathtt{x{:}S} \vdash \mathtt{t} : \mathtt{T}$. Moreover, the latter derivation has the same depth as the former.

Lemma [Permutation]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \mathtt{t} : \mathtt{T}$. Moreover, the latter derivation has the same depth as the former.

Fill in the missing parts of the proof of the substitution lemma on the following page.

- In the T-SUCC case, you need to fill in *both* the assumptions coming from the case analysis (the three blank lines at the beginning of the case) *and* the body of the argument.
- Your wording does not need to exactly match what is in the book or lecture notes, but every step required in the proof (use of an assumption, application of a lemma, use of the induction hypothesis, or use of a typing rule) must be mentioned explicitly.
- The cases for application, zero, and predecessor are omitted; you don't need to worry about these.

Lemma [Substitution]: If $\Gamma, \mathtt{x{:}S} \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{s} : \mathtt{S}$, then $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} : \mathtt{T}$.

Proof: By induction on the depth of a derivation of $\Gamma, \mathtt{x{:}S} \vdash \mathtt{t} : \mathtt{T}$. Proceed by cases on the final typing rule used in the derivation.

Case T-VAR: $\mathtt{t} = \mathtt{z}$
with $\mathtt{z{:}T} \in (\Gamma, \mathtt{x{:}S})$

Answer: There are two sub-cases to consider, depending on whether z *is* x *or another variable. If* $z = x$*, then* $[x \mapsto s]z = s$*. The required result is then* $\Gamma \vdash s : S$*, which is among the assumptions of the lemma. Otherwise,* $[x \mapsto s]z = z$*, and the desired result is immediate.*

Case T-ABS: $\mathtt{t} = \lambda \mathtt{y{:}T_2}.\mathtt{t_1}$ $\quad \mathtt{T} = \mathtt{T_2} \rightarrow \mathtt{T_1}$
$\Gamma, \mathtt{x{:}S}, \mathtt{y{:}T_2} \vdash \mathtt{t_1} : \mathtt{T_1}$

By our conventions on choice of bound variable names, we may assume $\mathtt{x} \neq \mathtt{y}$ and $\mathtt{y} \notin \mathit{FV}(\mathtt{s})$.
Answer: Using permutation *on the given subderivation, we obtain* Γ, $y{:}T_2$, $x{:}S \vdash t_1 : T_1$*. Using* weakening *on the other given derivation (*$\Gamma \vdash s : S$*), we obtain* Γ, $y{:}T_2 \vdash s : S$*. Now, by the induction hypothesis,* Γ, $y{:}T_2 \vdash [x \mapsto s]t_1 : T_1$*. By* T-ABS, $\Gamma \vdash \lambda y{:}T_2.\ [x \mapsto s]t_1 : T_2 \rightarrow T_1$*, i.e. (by the definition of substitution),* $\Gamma \vdash [x \mapsto s]\lambda y{:}T_2.\ t_1 : T_2 \rightarrow T_1$*.*

Case T-SUCC: $\mathtt{t} = \mathtt{succ\ t_1}$
$\Gamma, \mathtt{x : S} \vdash \mathtt{t_1 : Nat}$
$\mathtt{T} = \mathtt{Nat}$

Answer: By the induction hypothesis, $\Gamma \vdash [x \mapsto s]t_1$ *: Nat. By* T-SUCC, $\Gamma \vdash$ *succ* $([x \mapsto s]t_1)$ *: Nat, i.e.,* $\Gamma \vdash [x \mapsto s](succ\ t_1)$ *: Nat.*

Grading scheme: 6 points for each part. -1 for a minor omission (e.g., showing the result of a step but omitting the justification); -2 for omitting a step completely.

For reference: Boolean and arithmetic expressions

Syntax

t ::=		*terms*
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
	`0`	*constant zero*
	`succ t`	*successor*
	`pred t`	*predecessor*
	`iszero t`	*zero test*
v ::=		*values*
	`true`	*true value*
	`false`	*false value*
	`nv`	*numeric value*
nv ::=		*numeric values*
	`0`	*zero value*
	`succ nv`	*successor value*
T ::=		*types*
	`Bool`	*type of booleans*
	`Nat`	*type of numbers*

Evaluation

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \text{(E-IfTrue)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(E-If)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ } t_1 \longrightarrow \texttt{succ } t_1'} \qquad \text{(E-Succ)}$$

$$\texttt{pred 0} \longrightarrow \texttt{0} \qquad \text{(E-PredZero)}$$

$$\texttt{pred (succ } nv_1\texttt{)} \longrightarrow nv_1 \qquad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred } t_1 \longrightarrow \texttt{pred } t_1'} \qquad \text{(E-Pred)}$$

$$\texttt{iszero 0} \longrightarrow \texttt{true} \qquad \text{(E-IszeroZero)}$$

$$\texttt{iszero (succ } nv_1\texttt{)} \longrightarrow \texttt{false} \qquad \text{(E-IszeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero } t_1 \longrightarrow \texttt{iszero } t_1'} \qquad \text{(E-IsZero)}$$

continued on next page...

Typing

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\frac{\texttt{t}_1 : \texttt{Bool} \quad \texttt{t}_2 : \texttt{T} \quad \texttt{t}_3 : \texttt{T}}{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 : \texttt{T}} \qquad \text{(T-If)}$$

$$\texttt{0} : \texttt{Nat} \qquad \text{(T-Zero)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{succ t}_1 : \texttt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{pred t}_1 : \texttt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{iszero t}_1 : \texttt{Bool}} \qquad \text{(T-IsZero)}$$

For reference: Untyped lambda calculus

Syntax

$\mathtt{t}$ ::=		*terms*
	$\mathtt{x}$	*variable*
	$\lambda\mathtt{x.t}$	*abstraction*
	$\mathtt{t\ t}$	*application*
$\mathtt{v}$ ::=		*values*
	$\lambda\mathtt{x.t}$	*abstraction value*

Evaluation

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

$$(\lambda\mathtt{x.t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

For reference: Simply typed lambda-calculus with numbers

Syntax

t ::=		*terms*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	0	*constant zero*
	succ t	*successor*
	pred t	*predecessor*
v ::=		*values*
	λx:T.t	*abstraction value*
	nv	*numeric value*
nv ::=		*numeric values*
	0	*zero value*
	succ nv	*successor value*
T ::=		*types*
	Nat	*type of numbers*
	T→T	*type of functions*

Evaluation

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x{:}T_1.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ}\ t_1 \longrightarrow \texttt{succ}\ t_1'} \quad \text{(E-Succ)}$$

$$\texttt{pred}\ 0 \longrightarrow 0 \quad \text{(E-PredZero)}$$

$$\texttt{pred}\ (\texttt{succ}\ nv_1) \longrightarrow nv_1 \quad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred}\ t_1 \longrightarrow \texttt{pred}\ t_1'} \quad \text{(E-Pred)}$$

continued on next page...

Typing

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}:\mathtt{T}_1 . \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1 \ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-App)}$$

$$\Gamma \vdash \mathtt{0} : \mathtt{Nat} \qquad \text{(T-Zero)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{succ}\ \mathtt{t}_1 : \mathtt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{pred}\ \mathtt{t}_1 : \mathtt{Nat}} \qquad \text{(T-Pred)}$$