

Formal Type Theory (582655)
Separate Exam, March 26th, 2010
Sample solutions
Lauri Alanko

Answer all questions. You are free to provide auxiliary definitions in your solutions. Partial credit is given: if you do not know the full and correct solution, answer to the best of your knowledge.

Except where otherwise noted, all the questions concern a simply typed base language with functions, booleans, numbers and general recursion. Its Coq definition is given in Appendix A. Below, this language is simply called "the base language".

In your answers, you are free to use either Coq notation or the notation in TAPL, as you prefer.

Good luck!

1 (8 p)

In the following questions, note that "there is none" is also a valid and possible answer.

1.1 (2 p)

Give a type T such that $x : Bool \vdash \text{iszero}(\text{succ } 'x) : T$

Sample solution

There is no such type.

1.2 (2 p)

Give a typing context Γ such that $\Gamma \vdash \text{pred}('x \$ 'y) : Nat$

Sample solution

$\Gamma = \{ x : Bool \rightarrow Nat, y : Bool \}$

The type of y can be arbitrary as long as it is the domain of x .

1.3 (2 p)

Give a term t such that $t \longrightarrow \text{false}$

Sample solution

$t = \text{iszero}(\text{succ zero})$

1.4 (2 p)

Give a value v such that $(\lambda x : Nat, \text{if true then } 'x \text{ else zero}) \longrightarrow^* v$.

Sample solution

$v = (\lambda x : \text{Nat}, \text{if true then } 'x \text{ else zero})$

A common mistake was to forget that the many-step evaluation relation \longrightarrow^* is reflexive, i.e. $t \longrightarrow^* t$ for all t .

2 (12 p)

Each of the following questions presents some modifications to the base language. For each modification, consider the properties of *determinacy of evaluation* ("determinism"), *progress of well-typed terms* ("progress") and *preservation of typing by evaluation* ("type preservation"). For each property that *fails* in the presence of the modification, give a counterexample demonstrating its failure.

The questions below are not cumulative: each question presents modifications to the *base language*.

2.1 (3 p)

Suppose we add the following rule to the *eval* relation:

$$\begin{array}{l} | E_IfFunny : \text{forall } t1\ t2\ t3\ t3' : \text{term}, \\ \quad \text{eval } t3\ t3' \rightarrow \\ \quad \text{eval } (TmIf\ t1\ t2\ t3)\ (TmIf\ t1\ t2\ t3'). \end{array}$$

Now which properties fail?

Sample solution

Determinism fails. By *E_IfTrue* we have:

$\text{if true then false else iszero zero} \longrightarrow \text{false}$

But by *E_IfFunny* and *E_IsZeroZero* we also have:

$\text{if true then false else iszero zero} \longrightarrow \text{if true then false else true}$

2.2 (3 p)

Suppose we remove the *E_Pred* rule from the *eval* relation. Now which properties fail?

Sample solution

Progress fails. By *T_Pred* and *T_Zero* we have $\vdash \text{pred } (\text{pred zero}) : \text{Nat}$ but the term is not a value and cannot be evaluated further.

2.3 (3 p)

Suppose we remove the *T_False* rule from the *typing* relation. Now which properties fail?

Sample solution

Type preservation fails. We have $\vdash \text{iszero}(\text{succ zero}) : \text{Bool}$ and $\text{iszero}(\text{succ zero}) \longrightarrow \text{false}$, but false is no longer well-typed.

2.4 (3 p)

Suppose we add the following rule to the *typing* relation:

$$\begin{array}{l} | T_IfFunny : \text{forall } \Gamma \ t2 \ t3 \ T2 \ T3, \\ \quad \text{typing } \Gamma \ t2 \ T2 \rightarrow \\ \quad \text{typing } \Gamma \ t3 \ T3 \rightarrow \\ \quad \text{typing } \Gamma \ (TmIf \ TmTrue \ t2 \ t3) \ T2. \end{array}$$

Now which properties fail?

Sample solution

None of the properties fail. In particular, type preservation still holds: if a term is typed by the *T_IfFunny* rule, then the typing is of the form $\Gamma \vdash \text{if true then } t2 \text{ else } t3 : T2$, and this typing requires $\Gamma \vdash t2 : T2$ to also hold. The only way that the term can be evaluated is by the *T_IfTrue* rule: $\text{if true then } t2 \text{ else } t3 \longrightarrow t2$. Since $\Gamma \vdash t2 : T2$, type preservation holds.

3 (12 p)

3.1 (6 p)

The base language is not normalizing, i.e. not every well-typed term is normalizable. What, minimally, needs to be changed in order to make the language normalizing? Provide a one- or two-sentence rationale, but a full proof is not needed.

Sample solution

The language is not normalizing because of the general recursion operation *fixx*, or *TmFix*. It can be used to form non-normalizing terms like *fixx* $(\lambda f : \text{Nat} \rightarrow \text{Nat}, f)$. Removing the *fixx* operation (and associated evaluation and typing rules) makes the language normalizing.

In the answers, there were various interpretations of "minimally", but anything that removed or altered *fixx* while leaving the rest of the language untouched was accepted.

3.2 (6 p)

Suppose that we have changed the base language so that it is normalizing. Prove informally the following statement: every well-typed term will evaluate into a value in a finite number of steps. You can assume that the progress and type preservation properties hold.

Sample solution

Suppose t is well-typed, i.e. there is some T such that $\vdash t : T$.

Because t is well-typed and the language is normalizing, there is some normal form t' such that $t \longrightarrow^* t'$.

Because single-step evaluation preserves typing, a simple induction on the length of the evaluation (or the structure of the proof of $t \longrightarrow^* t'$) shows that $\vdash t' : T$.

Because t' is well-typed, then by the progress theorem, either t' is a value or there is some t'' such that $t' \longrightarrow t''$.

Because t' is a normal form, there is no t'' such that $t' \longrightarrow t''$.

Hence t' must be a value. □

4 (8 p)

Consider extending the base language with a family of list types, written $List\ T$, where T is the type of the elements of the list. The full definition of lists as found in TAPL is presented in Appendix B. From the definition it can be seen that values of type $List\ T$ have the form $cons\ [T]\ v1\ (cons\ [T]\ v2\ \dots\ (cons\ [T]\ vn\ (nil\ [T]))\ \dots)$.

4.1 (4 p)

Define a term $mapnat$ that has the following properties:

$$\vdash mapnat : (Nat \rightarrow Nat) \rightarrow List\ Nat \rightarrow List\ Nat$$
$$mapnat\ f\ (nil\ [Nat]) \longrightarrow^* nil\ [Nat]$$
$$mapnat\ f\ (cons\ [Nat]\ v1\ v2) \longrightarrow^* cons\ [Nat]\ (f\ v1)\ (mapnat\ f\ v2)$$

Sample solution

The term $mapnat$ can be defined as follows:

$$\begin{aligned} mapnat = & \lambda f : Nat \rightarrow Nat, fixx\ (\lambda rec : List\ Nat \rightarrow List\ Nat, \lambda l : List\ Nat, \\ & \text{if } isnil[Nat]\ l \\ & \text{then } nil[Nat] \\ & \text{else } cons[Nat]\ (f\ \$\ (head[Nat]\ l))\ (rec\ \$\ tail[Nat]\ l)) \end{aligned}$$

There was a minor ambiguity in the problem statement: the above properties must hold for any *value* f , not for any arbitrary term f . Thankfully this did not seem to confuse anyone.

Some answers took liberties interpreting what "defining a term" means, and defined a language extension where the above properties were simply taken at face value as evaluation and typing rules. This was penalized.

4.2 (4 p)

Now consider further extending the language with an extremely simple version of *list comprehensions* of the form $\{ t1 : T1 \mid x : T2 \leftarrow t2 \}$. Informally, the semantics of list comprehensions

is as follows: the comprehension term evaluates to a list whose elements are obtained by evaluating $t1$ once for each element of $t2$ (which should be a list of type $List\ T2$). Each time when evaluating $t1$, the variable x is substituted with the given element of $t2$. The elements in the resulting list are in the same order as the corresponding elements of $t2$. The term $t1$ should have the type T .

Here are examples of how list comprehension terms should be evaluated:

$$\{\text{iszero } 'x : Bool \mid x : Nat \leftarrow \text{cons } [Nat] (\text{succ zero}) (\text{cons } [Nat] \text{ zero } (\text{nil } [Nat]))\} \\ \longrightarrow^* \text{cons } [Bool] \text{ false } (\text{cons } [Bool] \text{ true } (\text{nil } [Bool]))$$

$$\{\text{succ } 'y : Nat \mid y : Nat \leftarrow \text{cons } [Nat] \text{ zero } (\text{cons } [Nat] (\text{succ zero}) (\text{nil } [Nat]))\} \\ \longrightarrow^* \text{cons } [Nat] (\text{succ zero}) (\text{cons } [Nat] (\text{succ } (\text{succ zero})) (\text{nil } [Nat]))$$

$$\{\text{if } 'z \text{ then succ zero else zero } : Nat \mid z : Bool \leftarrow \text{cons } [Bool] \text{ true } (\text{cons } [Bool] \text{ false } (\text{nil } [Bool]))\} \\ \longrightarrow^* \text{cons } [Nat] (\text{succ zero}) (\text{cons } [Nat] \text{ zero } (\text{nil } [Nat]))$$

Formalize the above semantics. You can either define the list comprehensions as a derived form by providing a translation into a term that does not contain list comprehensions, or you can write down explicit evaluation and typing rules that formalize the above semantics while retaining the progress, preservation and determinism properties of the language.

Sample solution

Perhaps the simplest solution is to define list comprehensions as a derived form that uses a recursive function much like *mapnat* above:

$$\{t1 : T1 \mid x : T2 \leftarrow t2\} = \\ (\lambda f : T2 \rightarrow T1, \text{fixx } (\lambda \text{rec} : List\ T2 \rightarrow List\ T1, \lambda l : List\ T2, \\ \text{if isnil}[T2]\ l \\ \text{then nil}[T1] \\ \text{else cons}[T1] (f \$ \text{head}[T2]\ l) (\text{rec } \$ \text{tail}[T2]\ l))) \\ \$ (\lambda x : T2, t1) \$ t2$$

Alternatively, we can extend the language with new terms and rules:

$$t ::= \dots \\ \mid \{t_1 : T_1 \mid x : T_2 \leftarrow t_2\}$$

$$\{t_1 : T_1 \mid x : T_2 \leftarrow \text{nil}[T_2]\} \rightarrow \text{nil}[T_1] \quad (\text{E-LCOMPNIL})$$

$$\{t_1 : T_1 \mid x : T_2 \leftarrow \text{cons}[T_2]v_1v_2\} \rightarrow \text{cons}[T_1]([x \mapsto v_1]t_1)\{t_1 : T_1 \mid x : T_2 \leftarrow v_2\} \quad (\text{E-LCOMPCONS})$$

$$\frac{t_2 \rightarrow t'_2}{\{t_1 : T_1 \mid x : T_2 \leftarrow t_2\} \rightarrow \{t_1 : T_1 \mid x : T_2 \leftarrow t'_2\}} \quad (\text{E-LCOMP})$$

$$\frac{\Gamma \vdash t_2 : List[T_2] \quad \Gamma, x : T_2 \vdash t_1 : T_1}{\Gamma \vdash \{t_1 : T_1 \mid x : T_2 \leftarrow t_2\} : List[T_1]} \quad (\text{T-LCOMP})$$

A Simply typed lambda calculus with booleans, numbers and general recursion

Definition $eq_var := beq_nat$.

Definition $label := nat$.

Definition $index := nat$.

Definition $var := nat$.

Inductive $type : Set :=$

| $TyBool : type$

| $TyNat : type$

| $TyArrow : type \rightarrow type \rightarrow type$.

Inductive $ctx : Set :=$

| $CtxEmpty : ctx$

| $CtxExtend : ctx \rightarrow var \rightarrow type \rightarrow ctx$.

Inductive $term : Set :=$

| $TmTrue : term$

| $TmFalse : term$

| $TmIf : term \rightarrow term \rightarrow term \rightarrow term$

| $TmZero : term$

| $TmSucc : term \rightarrow term$

| $TmPred : term \rightarrow term$

| $TmIszero : term \rightarrow term$

| $TmVar : var \rightarrow term$

| $TmAbs : var \rightarrow type \rightarrow term \rightarrow term$

| $TmApp : term \rightarrow term \rightarrow term$

| $TmFix : term \rightarrow term$.

Notation $''true'' := (TmTrue) : term_scope$.

Notation $''false'' := (TmFalse) : term_scope$.

Notation $''if' t1 'then' t2 'else' t3'' :=$

$((TmIf t1 t2 t3)) : term_scope$.

Notation $''Bool'' := TyBool$ (at level 0, no associativity) : ty_scope .

Notation $''succ' t'' := ((TmSucc t)) : term_scope$.

Notation $''pred' t'' := ((TmPred t)) : term_scope$.

Notation $''0'' := (TmZero) : term_scope$.

Notation $''iszero' t'' := ((TmIszero t)) : term_scope$.

Notation $''Nat'' := TyNat$ (at level 0, no associativity) : ty_scope .

Notation $''x : T, t'' := (TmAbs x T t) : term_scope$.

Notation $''t1 \$ t2'' := (TmApp t1 t2) : term_scope$.

Notation $''n'' := (TmVar n)$ (at level 7, no associativity) : $term_scope$.

Notation $''T \rightarrow U'' := (TyArrow T U) : ty_scope$.

Notation $''fixx' t'' := ((TmFix t)) : term_scope$.

Fixpoint $subst x t' t :=$

match t with

| $(TmVar y) \Rightarrow$ if $beq_nat x y$ then t' else t

```

| (t1 $ t2)%tm ⇒ (subst x t' t1 $ subst x t' t2)%tm
| (" y : T , t1)%tm ⇒ if beq_nat x y then t else (" y : T , subst x t' t1)%tm
| (fixx t1)%tm ⇒ (fixx (subst x t' t1))%tm
| (if t1 then t2 else t3)%tm ⇒
  (if (subst x t' t1) then (subst x t' t2) else (subst x t' t3))%tm
| (succ t1)%tm ⇒ (succ (subst x t' t1))%tm
| (pred t1)%tm ⇒ (pred (subst x t' t1))%tm
| (iszero t1)%tm ⇒ (iszero (subst x t' t1))%tm
| _ ⇒ t

```

end.

Inductive bound : var → type → ctx → Prop :=

```

| B_Here : forall (x:var) (T:type) (Γ:ctx),
  bound x T (CtxExtend Γ x T)
| B_Next : forall (x:var) (T:type) (Γ:ctx) (y:var) (T':type),
  (Logic.not (x = y)) →
  bound x T Γ →
  bound x T (CtxExtend Γ y T')

```

with fvs : term → var → Prop :=

```

| FVS_If1 : forall (t1 t2 t3:term) (x:var),
  fvs t1 x →
  fvs (TmIf t1 t2 t3) x
| FVS_If2 : forall (t1 t2 t3:term) (x:var),
  fvs t2 x →
  fvs (TmIf t1 t2 t3) x
| FVS_If3 : forall (t1 t2 t3:term) (x:var),
  fvs t3 x →
  fvs (TmIf t1 t2 t3) x
| FVS_Var : forall (x:var),
  fvs (TmVar x) x
| FVS_Abs : forall (y:var) (T:type) (t:term) (x:var),
  (Logic.not (x = y)) →
  fvs t x →
  fvs (TmAbs y T t) x
| FVS_App1 : forall (t1 t2:term) (x:var),
  fvs t1 x →
  fvs (TmApp t1 t2) x
| FVS_App2 : forall (t1 t2:term) (x:var),
  fvs t2 x →
  fvs (TmApp t1 t2) x

```

with typing : ctx → term → type → Prop :=

```

| T_True : forall (Γ:ctx),
  typing Γ TmTrue TyBool
| T_False : forall (Γ:ctx),
  typing Γ TmFalse TyBool
| T_If : forall (Γ:ctx) (t1 t2 t3:term) (T:type),
  typing Γ t1 TyBool →

```

```

    typing  $\Gamma$   $t_2$   $T \rightarrow$ 
    typing  $\Gamma$   $t_3$   $T \rightarrow$ 
    typing  $\Gamma$  (TmIf  $t_1$   $t_2$   $t_3$ )  $T$ 
| T_Zero : forall ( $\Gamma$ :ctx),
    typing  $\Gamma$  TmZero TyNat
| T_Succ : forall ( $\Gamma$ :ctx) ( $t_1$ :term),
    typing  $\Gamma$   $t_1$  TyNat  $\rightarrow$ 
    typing  $\Gamma$  (TmSucc  $t_1$ ) TyNat
| T_Pred : forall ( $\Gamma$ :ctx) ( $t_1$ :term),
    typing  $\Gamma$   $t_1$  TyNat  $\rightarrow$ 
    typing  $\Gamma$  (TmPred  $t_1$ ) TyNat
| T_IsZero : forall ( $\Gamma$ :ctx) ( $t_1$ :term),
    typing  $\Gamma$   $t_1$  TyNat  $\rightarrow$ 
    typing  $\Gamma$  (TmIszero  $t_1$ ) TyBool
| T_Var : forall ( $\Gamma$ :ctx) ( $x$ :var) ( $T$ :type),
    bound  $x$   $T$   $\Gamma \rightarrow$ 
    typing  $\Gamma$  (TmVar  $x$ )  $T$ 
| T_Abs : forall ( $\Gamma$ :ctx) ( $x$ :var) ( $T_1$ :type) ( $t_2$ :term) ( $T_2$ :type),
    typing (CtxExtend  $\Gamma$   $x$   $T_1$ )  $t_2$   $T_2 \rightarrow$ 
    typing  $\Gamma$  (TmAbs  $x$   $T_1$   $t_2$ ) (TyArrow  $T_1$   $T_2$ )
| T_App : forall ( $\Gamma$ :ctx) ( $t_1$   $t_2$ :term) ( $T_{12}$   $T_{11}$ :type),
    typing  $\Gamma$   $t_1$  (TyArrow  $T_{11}$   $T_{12}$ )  $\rightarrow$ 
    typing  $\Gamma$   $t_2$   $T_{11} \rightarrow$ 
    typing  $\Gamma$  (TmApp  $t_1$   $t_2$ )  $T_{12}$ 
| T_Fix : forall ( $\Gamma$ :ctx) ( $t_1$ :term) ( $T_1$ :type),
    typing  $\Gamma$   $t_1$  (TyArrow  $T_1$   $T_1$ )  $\rightarrow$ 
    typing  $\Gamma$  (TmFix  $t_1$ )  $T_1$ .

```

Inductive $nv : term \rightarrow Prop :=$

```

| NV_Zero :
    nv TmZero
| NV_Succ : forall ( $t$ :term),
    nv  $t \rightarrow$ 
    nv (TmSucc  $t$ )

```

with $val : term \rightarrow Prop :=$

```

| Val_True :
    val TmTrue
| Val_False :
    val TmFalse
| Val_NV : forall ( $t$ :term),
    nv  $t \rightarrow$ 
    val  $t$ 
| Val_Abs : forall ( $x$ :var) ( $T$ :type) ( $t$ :term),
    val (TmAbs  $x$   $T$   $t$ )

```

with $eval : term \rightarrow term \rightarrow Prop :=$

```

| E_IfTrue : forall ( $t_2$   $t_3$ :term),
    eval (TmIf TmTrue  $t_2$   $t_3$ )  $t_2$ 

```

$| E_IfFalse : \text{forall } (t2\ t3:\text{term}),$
 $\quad \text{eval } (TmIf\ TmFalse\ t2\ t3)\ t3$

$| E_If : \text{forall } (t1\ t2\ t3\ t1':\text{term}),$
 $\quad \text{eval } t1\ t1' \rightarrow$
 $\quad \text{eval } (TmIf\ t1\ t2\ t3)\ (TmIf\ t1'\ t2\ t3)$

$| E_Succ : \text{forall } (t1\ t1':\text{term}),$
 $\quad \text{eval } t1\ t1' \rightarrow$
 $\quad \text{eval } (TmSucc\ t1)\ (TmSucc\ t1')$

$| E_PredZero :$
 $\quad \text{eval } (TmPred\ TmZero)\ TmZero$

$| E_PredSucc : \text{forall } (t1:\text{term}),$
 $\quad \text{nv } t1 \rightarrow$
 $\quad \text{eval } (TmPred\ (TmSucc\ t1))\ t1$

$| E_Pred : \text{forall } (t1\ t1':\text{term}),$
 $\quad \text{eval } t1\ t1' \rightarrow$
 $\quad \text{eval } (TmPred\ t1)\ (TmPred\ t1')$

$| E_IsZeroZero :$
 $\quad \text{eval } (TmIszero\ TmZero)\ TmTrue$

$| E_IsZeroSucc : \text{forall } (t1:\text{term}),$
 $\quad \text{nv } t1 \rightarrow$
 $\quad \text{eval } (TmIszero\ (TmSucc\ t1))\ TmFalse$

$| E_IsZero : \text{forall } (t1\ t1':\text{term}),$
 $\quad \text{eval } t1\ t1' \rightarrow$
 $\quad \text{eval } (TmIszero\ t1)\ (TmIszero\ t1')$

$| E_App1 : \text{forall } (t1\ t2\ t1':\text{term}),$
 $\quad \text{eval } t1\ t1' \rightarrow$
 $\quad \text{eval } (TmApp\ t1\ t2)\ (TmApp\ t1'\ t2)$

$| E_App2 : \text{forall } (t1\ t2\ t2':\text{term}),$
 $\quad \text{val } t1 \rightarrow$
 $\quad \text{eval } t2\ t2' \rightarrow$
 $\quad \text{eval } (TmApp\ t1\ t2)\ (TmApp\ t1\ t2')$

$| E_AppAbs : \text{forall } (x:\text{var})\ (T:\text{type})\ (t12\ t2:\text{term}),$
 $\quad \text{val } t2 \rightarrow$
 $\quad \text{eval } (TmApp\ (TmAbs\ x\ T\ t12)\ t2)\ (\text{subst } x\ t2\ t12)$

$| E_FixBeta : \text{forall } (x:\text{var})\ (T1:\text{type})\ (t2:\text{term}),$
 $\quad \text{eval } (TmFix\ (TmAbs\ x\ T1\ t2))\ (\text{subst } x\ (TmFix\ (TmAbs\ x\ T1\ t2))\ t2)$

$| E_Fix : \text{forall } (t1\ t1':\text{term}),$
 $\quad \text{eval } t1\ t1' \rightarrow$
 $\quad \text{eval } (TmFix\ t1)\ (TmFix\ t1')$

Notation " $t1 \longrightarrow t2$ " := $(\text{eval } t1\ t2) : \text{type_scope}$.

Notation " $\{\}$ " := $\text{CtxEmpty} : \text{ctx_scope}$.

Notation " $\{ G, x : T \}$ " := $((\text{CtxExtend } \Gamma\ x\ T)) : \text{ctx_scope}$.

Notation " $\vdash t : T$ " := $(\text{typing } \text{CtxEmpty}\ t\ T) : \text{type_scope}$.

Notation " $G \vdash t : T$ " := $(\text{typing } \Gamma\ t\ T) : \text{type_scope}$.

Notation " $[x \longrightarrow_i t1] t2$ " := $(\text{subst } x\ t1\ t2) : \text{term_scope}$.

B Lists

$\rightarrow \mathbb{B}$ List	<i>Extends λ_- (9-1) with booleans (8-1)</i>
<p><i>New syntactic forms</i></p> <p>$t ::= \dots$</p> <ul style="list-style-type: none"> $\text{nil}[T]$ <i>terms:</i> empty list $\text{cons}[T] \ t \ t$ <i>list constructor</i> $\text{isnil}[T] \ t$ <i>test for empty list</i> $\text{head}[T] \ t$ <i>head of a list</i> $\text{tail}[T] \ t$ <i>tail of a list</i> <p>$v ::= \dots$</p> <ul style="list-style-type: none"> $\text{nil}[T]$ <i>values:</i> empty list $\text{cons}[T] \ v \ v$ <i>list constructor</i> <p>$T ::= \dots$</p> <ul style="list-style-type: none"> $\text{List } T$ <i>types:</i> type of lists 	<div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\frac{t_1 \rightarrow t'_1}{\text{isnil}[T] \ t_1 \rightarrow \text{isnil}[T] \ t'_1} \quad (\text{E-ISNIL})$ </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\text{head}[S] \ (\text{cons}[T] \ v_1 \ v_2) \rightarrow v_1 \quad (\text{E-HEADCONS})$ </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\frac{t_1 \rightarrow t'_1}{\text{head}[T] \ t_1 \rightarrow \text{head}[T] \ t'_1} \quad (\text{E-HEAD})$ </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\text{tail}[S] \ (\text{cons}[T] \ v_1 \ v_2) \rightarrow v_2 \quad (\text{E-TAILCONS})$ </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\frac{t_1 \rightarrow t'_1}{\text{tail}[T] \ t_1 \rightarrow \text{tail}[T] \ t'_1} \quad (\text{E-TAIL})$ </div> <p><i>New typing rules</i> $\Gamma \vdash t : T$</p> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\Gamma \vdash \text{nil} [T_1] : \text{List } T_1 \quad (\text{T-NIL})$ </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : \text{List } T_1}{\Gamma \vdash \text{cons}[T_1] \ t_1 \ t_2 : \text{List } T_1} \quad (\text{T-CONS})$ </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\frac{\Gamma \vdash t_1 : \text{List } T_{11}}{\Gamma \vdash \text{isnil}[T_{11}] \ t_1 : \text{Bool}} \quad (\text{T-ISNIL})$ </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\frac{\Gamma \vdash t_1 : \text{List } T_{11}}{\Gamma \vdash \text{head}[T_{11}] \ t_1 : T_{11}} \quad (\text{T-HEAD})$ </div> <div style="border: 1px solid black; padding: 5px;"> $\frac{\Gamma \vdash t_1 : \text{List } T_{11}}{\Gamma \vdash \text{tail}[T_{11}] \ t_1 : \text{List } T_{11}} \quad (\text{T-TAIL})$ </div>
<p><i>New evaluation rules</i> $t \rightarrow t'$</p> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\frac{t_1 \rightarrow t'_1}{\text{cons}[T] \ t_1 \ t_2 \rightarrow \text{cons}[T] \ t'_1 \ t_2} \quad (\text{E-CONS1})$ </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\frac{t_2 \rightarrow t'_2}{\text{cons}[T] \ v_1 \ t_2 \rightarrow \text{cons}[T] \ v_1 \ t'_2} \quad (\text{E-CONS2})$ </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> $\text{isnil}[S] \ (\text{nil}[T]) \rightarrow \text{true} \quad (\text{E-ISNILNIL})$ </div> <div style="border: 1px solid black; padding: 5px;"> $\text{isnil}[S] \ (\text{cons}[T] \ v_1 \ v_2) \rightarrow \text{false} \quad (\text{E-ISNILCONS})$ </div>	

Figure 11-13: Lists