Appendix

$$\frac{\begin{array}{c}\mathrm{p} \;=\; \mathrm{famdef}{*}\ e \qquad L = \{\mathtt{self} = \mathtt{prog},\ \mathtt{super} = \mathtt{null},\ \mathtt{NEST} = \{A \mapsto L' : \mathcal{P}(A, L')\}\} \\ \mathcal{P}(A, L') = \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \{\ldots\}\ \in\ \mathrm{famdef}{*}\ \wedge\ L' = \mathtt{parse}(\mathtt{prog}, \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \{\ldots\})\end{array}}{\mathtt{parse}(\mathrm{p}) \;=\; L}\ \text{(PARSE-PROG)}$$

$$\frac{\begin{array}{c}L = \{\mathtt{self} = \mathtt{self}(sp.A),\ \mathtt{super} = a.A',\ \mathtt{NEST},\ \mathtt{TYPES},\ \mathtt{DEFS},\ \mathtt{ADTS},\ \mathtt{FUNS},\ \mathtt{CASES}\} \\ \mathtt{NEST} = \{A' \mapsto L' : \mathcal{P}(A', L')\} \\ \mathcal{P}(A', L') = \textbf{Family}\ A'\ (\textbf{extends}\ a'.A'')?\ \{\ldots\}\ \in\ \mathrm{famdef}{*}\ \wedge\ L' = \mathtt{parse}(\mathtt{prog}, \textbf{Family}\ A'\ (\textbf{extends}\ a'.A'')?\ \{\ldots\}) \\ \mathtt{TYPES} = \{R_{((+)?=)} \mapsto \{(f_i : T_i){*}\} : \textbf{type}\ R\ (+)?{=}\ \{(f_i : T_i (= v_i)?){*}\}\ \in\ \mathrm{typedef}{*}\} \\ \mathtt{DEFS} = \{R_{((+)?=)} \mapsto \{(f_i = v_i){*}\} : \textbf{type}\ R\ (+)?{=}\ \{(f_i : T_i (= v_i)?){*}\}\ \in\ \mathrm{typedef}{*}\} \\ \mathtt{ADTS} = \{R_{((+)?=)} \mapsto \overline{C_j\ \{(f_i : T_i){*}\}} : \textbf{type}\ R\ (+)?{=}\ \overline{C_j\ \{(f_i : T_i){*}\}}\ \in\ \mathrm{adtdef}{*}\} \\ \mathtt{FUNS} = \{m \mapsto (T \to T', \lambda(x : T).e) : \textbf{val}\ m : T \to T' = \lambda(x : T).e\ \in\ \mathrm{fundef}{*}\} \\ \mathtt{CASES} = \{c_{((+)?=)} \mapsto (\langle a.R\rangle, T \to T', \lambda(x : T).e) : \textbf{cases}\ c\ \langle a.R\rangle : T \to T'\ (+)?{=}\ \lambda(x : T).e\ \in\ \mathrm{casesdef}{*}\}\end{array}}{\mathtt{parse}(sp,\ \textbf{Family}\ A\ (\textbf{extends}\ a.A')?\ \{\mathrm{famdef}{*}\ \ \mathrm{typdef}{*}\ \ \mathrm{adtdef}{*}\ \ \mathrm{fundef}{*}\ \ \mathrm{casesdef}{*}\}) = L}\ \text{(PARSE-FAMDEF)}$$

Figure 14. Parsing programs and family definitions.

$$\left\{\begin{array}{c}\mathtt{self} = sp \\ \mathtt{super} = a \\ \mathtt{NEST} = \{(A_n \mapsto L_n){*}\} \\ \mathtt{TYPES} = \{(R_{q\ ((+)?=)} \mapsto \{(f_i : T_i){*}\}){*}\} \\ \mathtt{DEFS} = \{(R_{q\ ((+)?=)} \mapsto \overline{\{(f_i = v_i){*}\}}){*}\} \\ \mathtt{ADTS} = \{(R_{u\ ((+)?=)} \mapsto \overline{C_j\ \{(f_i : T_i){*}\}}){*}\} \\ \mathtt{FUNS} = \{(m_w \mapsto (T_w \to T'_w,\ \lambda(x_w : T_w).e_w)){*}\} \\ \mathtt{CASES} = \{(c_{z\ ((+)?=)} \mapsto (\langle a_z.R_z\rangle,\ T_z \to T'_z,\ \lambda(x_z : T_z).e_z)){*}\}\end{array}\right\}$$

Figure 15. Linkage syntax.

$$\frac{\begin{array}{c}sp \notin \mathtt{ancestors}(sp) \\ \forall\, A \mapsto L\ \in\ \mathtt{NEST},\ \mathrm{sp} :: \mathrm{K} \vdash \mathtt{WF}(L) \\ \forall\, R \mapsto \{(f_i : T_i){*}\}\ \in\ \mathtt{TYPES},\ \mathrm{K} \vdash \mathtt{WF}(\{(f_i : T_i){*}\}) \\ \forall\, R \mapsto \{(f_i = v_i){*}\}\ \in\ \mathtt{DEFS},\ \exists\, R \mapsto \{(f'_i : T'_i){*}\}\ \in\ \mathtt{TYPES}\ \wedge\ \forall i,\ (f_i : T_i)\ \in\ (f'_i : T'_i){*}\ \wedge\ \mathrm{K}; [\,] \vdash v_i : T_i \\ \forall\, R \mapsto \overline{C_j\ \{(f_i : T_i){*}\}}\ \in\ \mathtt{ADTS},\ \forall j,\ \mathrm{K} \vdash \mathtt{WF}(\{(f_i : T_i){*}\}) \\ \forall\, m \mapsto (T \to T', \lambda(x : T).e)\ \in\ \mathtt{FUNS},\ \mathrm{K} \vdash \mathtt{WF}(T \to T')\ \wedge\ \mathrm{K}; [\,] \vdash \lambda(x : T).e : T \to T' \\ \forall\, c \mapsto (\langle a.R\rangle, T \to T', \lambda(x : T).e)\ \in\ \mathtt{CASES},\ \mathrm{K} \vdash \mathtt{WF}(T \to T')\ \wedge\ \mathrm{K}; [\,] \vdash \lambda(x : T).e : T \to T'\end{array}}{\mathrm{K} \vdash \mathtt{WF}(\{\mathtt{self} = sp,\ \mathtt{super} = a,\ \mathtt{NEST},\ \mathtt{TYPES},\ \mathtt{DEFS},\ \mathtt{ADTS},\ \mathtt{FUNS},\ \mathtt{CASES}\})}\ \text{(WF-Linkage)}$$

Figure 16. Linkage well-formedness.

$$\boxed{\mathrm{K} \vdash \mathtt{WF}(T)}$$

$$\frac{}{\mathrm{K} \vdash \mathtt{WF}(\mathrm{N})} \quad \text{(WF-Num)} \qquad \frac{}{\mathrm{K} \vdash \mathtt{WF}(\mathrm{B})} \quad \text{(WF-Bool)} \qquad \frac{\mathrm{K} \vdash \mathtt{WF}(T) \qquad \mathrm{K} \vdash \mathtt{WF}(T')}{\mathrm{K} \vdash \mathtt{WF}(T \to T')} \quad \text{(WF-Arrow)}$$

$$\frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad R = \{(f_i : T_i)*\} \ \in \ L.\mathrm{TYPES} \ \vee \ R = \overline{C_j \ \{(f_i : T_i)*\}} \ \in \ L.\mathrm{ADTS}}{\mathrm{K} \vdash \mathtt{WF}(a.R)} \quad \text{(WF-Named)}$$

$$\frac{\forall i, \ \mathrm{K} \vdash \mathtt{WF}(T_i) \qquad \forall i \ j, \ i \neq j \implies f_i \neq f_j}{\mathrm{K} \vdash \mathtt{WF}(\{(f_i : T_i)*\})} \quad \text{(WF-Record)}$$

Figure 17. Well-formedness of types.

$$\boxed{\mathrm{K} \vdash T \ <: \ T'}$$

$$\frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad R = \{(f_d : T_d)*\} \ \in \ L.\mathrm{TYPES} \qquad \mathrm{K} \vdash \{(f_d : T_d)*\} \ <: \ T'}{\mathrm{K} \vdash a.R \ <: \ T'} \quad \text{(Sub-Fam)} \qquad \frac{}{\mathrm{K} \vdash T \ <: \ T} \quad \text{(Sub-Refl)}$$

$$\frac{\mathrm{K} \vdash T \ <: \ S \qquad \mathrm{K} \vdash S' \ <: \ T'}{\mathrm{K} \vdash S \to S' \ <: \ T \to T'} \quad \text{(Sub-Fun)} \qquad \frac{\forall j, \ \exists T, \ \mathrm{K} \vdash T \ <: \ T_j \ \wedge \ (f_j : T) \ \in \ (f_i : T_i)*}{\mathrm{K} \vdash \{(f_i : T_i)*\} \ <: \ \{(f_j : T_j)*\}} \quad \text{(Sub-Rec)}$$

Figure 18. Subtyping relation.

$\boxed{\mathrm{K};\Gamma \vdash e : T}$

$$\frac{}{\mathrm{K};\Gamma \vdash n : \mathsf{N}}\ \text{(T-Num)} \qquad \frac{}{\mathrm{K};\Gamma \vdash b : \mathsf{B}}\ \text{(T-Bool)} \qquad \frac{x : T \ \in\ \Gamma}{\mathrm{K};\Gamma \vdash x : T}\ \text{(T-Var)}$$

$$\frac{\mathrm{K} \vdash \mathtt{WF}(T) \qquad \mathrm{K};(x : T, \Gamma) \vdash e : T'}{\mathrm{K};\Gamma \vdash \lambda(x : T).e : T \to T'}\ \text{(T-Lam)} \qquad \frac{\mathrm{K};\Gamma \vdash e : T \qquad \mathrm{K};\Gamma \vdash g : T \to T'}{\mathrm{K};\Gamma \vdash g\ e : T'}\ \text{(T-App)}$$

$$\frac{\forall i,\ \mathrm{K};\Gamma \vdash e_i : T_i}{\mathrm{K};\Gamma \vdash \{(f_i = e_i)*\} : \{(f_i : T_i)*\}}\ \text{(T-Rec)} \qquad \frac{\mathrm{K};\Gamma \vdash e : \{(f_i : T_i)*\} \qquad f : T \ \in\ (f_i : T_i)*}{\mathrm{K};\Gamma \vdash e.f : T}\ \text{(T-Proj)}$$

$$\frac{\mathrm{K};\Gamma \vdash e : T' \qquad \mathrm{K} \vdash T' <: T}{\mathrm{K};\Gamma \vdash e : T}\ \text{(T-Subs)} \qquad \frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad m : T \to T' = \lambda(x : T).e \ \in\ L.\mathrm{FUNS}}{\mathrm{K};\Gamma \vdash a.m : T \to T'}\ \text{(T-FamFun)}$$

$$\frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad R = \{(f_i : T_i)*\} \ \in\ L.\mathrm{TYPES} \qquad \forall i,\ \mathrm{K};\Gamma \vdash e_i : T_i}{\mathrm{K};\Gamma \vdash a.R(\{(f_i = e_i)*\}) : a.R}\ \text{(T-Constr)}$$

$$\frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad R = \overline{C_j\ \{(f_i : T_i)*\}} \ \in\ L.\mathrm{ADTS} \qquad C\ \{(f_k : T_k)*\} \ \in\ \overline{C_j\ \{(f_i : T_i)*\}} \qquad \forall k,\ \mathrm{K};\Gamma \vdash e_k : T_k}{\mathrm{K};\Gamma \vdash a.R(C\ \{(f_k = e_k)*\}) : a.R}\ \text{(T-ADT)}$$

$$\frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad c\ \langle a'.R\rangle : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\} = \lambda(x : \{(f_i : T_i)*\}).\{(C_j = \lambda(y_j : T_j).e_j)*\} \ \in\ L.\mathrm{CASES}}{\mathrm{K};\Gamma \vdash a.c : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\}}\ \text{(T-Cases)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash a' \rightsquigarrow L \qquad \mathrm{K};\Gamma \vdash e : a'.R \qquad R = \overline{C_j\ \{(f_i : T_i)*\}} \ \in\ L.\mathrm{ADTS} \\ \mathrm{K};\Gamma \vdash a.c : \{(f_{arg} : T_{arg})*\} \to \{(C_j : \{(f_i : T_i)*\} \to T)*\} \qquad \mathrm{K};\Gamma \vdash \{(f_{arg} = e_{arg})*\} : \{(f_{arg} : T_{arg})*\}\end{array}}{\mathrm{K};\Gamma \vdash \mathtt{match}\ e\ \mathtt{with}\ a.c\ \{(f_{arg} = e_{arg})*\} : T}\ \text{(T-Match)}$$

$$\frac{\mathrm{K};\Gamma \vdash e : \mathsf{B} \qquad \mathrm{K};\Gamma \vdash g : T \qquad \mathrm{K};\Gamma \vdash g' : T}{\mathrm{K};\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ g\ \mathtt{else}\ g' : T}\ \text{(T-If)}$$

$\boxed{\mathrm{K};\Gamma \vdash \mathrm{p} : T}$

$$\frac{\mathrm{p} = \mathrm{famdef}_i *\ e \qquad \forall i,\ [\mathrm{prog}] \vdash \mathtt{WF}(\mathrm{famdef}_i) \qquad [\mathrm{prog}];[\,] \vdash e : T}{[\,];[\,] \vdash \mathrm{p} : T}\ \text{(T-Prog)}$$

Figure 19. Typing rules for expressions and programs.

$$\boxed{\mathrm{K} \vdash e \longrightarrow e'}$$

$$\frac{\mathrm{K} \vdash g \longrightarrow g'}{\mathrm{K} \vdash g\ e \longrightarrow g'\ e}\ \text{(R-App)} \qquad \frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash v\ e \longrightarrow v\ e'}\ \text{(R-LamArg)} \qquad \frac{}{\mathrm{K} \vdash (\lambda(x : T).e)\ v \longrightarrow [x := v]\ e}\ \text{(R-LamApply)}$$

$$\frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash e.f \longrightarrow e'.f}\ \text{(R-Proj)} \qquad \frac{(f = v) \ \in\ (f_i = v_i)*}{\mathrm{K} \vdash \{(f_i = v_i)*\}.f \longrightarrow v}\ \text{(R-RecProj)} \qquad \frac{(f = v) \ \in\ (f_i = v_i)*}{\mathrm{K} \vdash a.R(\{(f_i = v_i)*\}).f \longrightarrow v}\ \text{(R-InstProj)}$$

$$\frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad m : T \to T' = \lambda(x : T).e \ \in\ L.\texttt{FUNS}}{\mathrm{K} \vdash a.m \longrightarrow \lambda(x : T).e}\ \text{(R-FamFun)} \qquad \frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash a.R(e) \longrightarrow a.R(e')}\ \text{(R-Instance)}$$

$$\frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash \texttt{if}\ e\ \texttt{then}\ g\ \texttt{else}\ g' \longrightarrow \texttt{if}\ e'\ \texttt{then}\ g\ \texttt{else}\ g'}\ \text{(R-IfGuard)} \qquad \frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash a.R(C\ e) \longrightarrow a.R(C\ e')}\ \text{(R-ADT)}$$

$$\frac{}{\mathrm{K} \vdash \texttt{if true then}\ g\ \texttt{else}\ g' \longrightarrow g}\ \text{(R-IfTrue)} \qquad \frac{}{\mathrm{K} \vdash \texttt{if false then}\ g\ \texttt{else}\ g' \longrightarrow g'}\ \text{(R-IfFalse)}$$

$$\frac{\mathrm{K} \vdash e_j \longrightarrow e'_j}{\mathrm{K} \vdash \{f_0 = v_0, ..., f_i = v_i, f_j = e_j, ...\} \longrightarrow \{f_0 = v_0, ..., f_i = v_i, f_j = e'_j, ...\}}\ \text{(R-Rec)}$$

$$\frac{\mathrm{K} \vdash e \longrightarrow e'}{\mathrm{K} \vdash \texttt{match}\ e\ \texttt{with}\ a.c\ \{(f_{arg} = e_{arg})*\} \longrightarrow \texttt{match}\ e'\ \texttt{with}\ a.c\ \{(f_{arg} = e_{arg})*\}}\ \text{(R-MatchExp)}$$

$$\frac{\mathrm{K} \vdash \{(f_{arg} = e_{arg})*\} \longrightarrow \{(f_{arg} = e'_{arg})*\}}{\mathrm{K} \vdash \texttt{match}\ v\ \texttt{with}\ a.c\ \{(f_{arg} = e_{arg})*\} \longrightarrow \texttt{match}\ v\ \texttt{with}\ a.c\ \{(f_{arg} = e'_{arg})*\}}\ \text{(R-MatchCases)}$$

$$\frac{}{\mathrm{K} \vdash \texttt{match}\ a'.R(C\ \{(f_k = v_k)*\})\ \texttt{with}\ a.c\ \{(f_{arg} = v_{arg})*\} \longrightarrow (a.c\ \{(f_{arg} = v_{arg})*\}).C\ \{(f_k = v_k)*\}}\ \text{(R-MatchFinal)}$$

$$\frac{\mathrm{K} \vdash a \rightsquigarrow L \qquad c\ \langle a'.R \rangle : \{(f_i : T_i)*\} \to \{(C_j : T_j \to T'_j)*\} = \lambda(x : \{(f_i : T_i)*\}).\{(C_j = \lambda(y_j : T_j).e_j)*\} \ \in\ L.\texttt{CASES}}{\mathrm{K} \vdash a.c \longrightarrow \lambda(x : \{(f_i : T_i)*\}).\{(C_j = \lambda(y_j : T_j).e_j)*\}}\ \text{(R-Cases)}$$

Figure 20. Reduction relation.

$\boxed{[x := s]\ e = e'}$

$$S_{Nat} : [x := s]\ n \ = \ n \qquad S_{Bool} : [x := s]\ b \ = \ b \qquad S_{Var} : [x := s]\ x \ = \ s$$

$$S_{VarNeq} : [x := s]\ y \ = \ y \qquad S_{LamNeq} : [x := s]\ \lambda(y : T).e \ = \ \lambda(y : T).([x := s]\ e)$$

$$S_{App} : [x := s]\ g\ e \ = \ ([x := s]\ g)([x := s]\ e) \qquad S_{Lam} : [x := s]\ \lambda(x : T).e \ = \ \lambda(x : T).e$$

$$S_{Rec} : [x := s]\ \{(f_i = e_i)*\} \ = \ \{(f_i = ([x := s]\ e_i))*\} \qquad S_{RecField} : [x := s]\ e.f \ = \ ([x := s]\ e).f$$

$$S_{FamFun} : [x := s]\ a.m \ = \ a.m \qquad S_{Constr} : [x := s]\ a.R(\{(f_i = e_i)*\}) \ = \ a.R(\{(f_i = ([x := s]\ e_i))*\})$$

$$S_{Cases} : [x := s]\ a.c \ = \ a.c \qquad S_{ADT} : [x := s]\ a.R(C\ \{(f_i = e_i)*\}) \ = \ a.R(C\ \{(f_i = ([x := s]\ e_i))*\})$$

$$S_{Match} : [x := s]\ \texttt{match}\ e\ \texttt{with}\ a.c\ \{(f_{arg} = e_{arg})*\} \ = \ \texttt{match}\ ([x := s]\ e)\ \texttt{with}\ a.c\ ([x := s]\ \{(f_{arg} = e_{arg})*\})$$

$$S_{If} : [x := s]\ \texttt{if}\ e\ \texttt{then}\ g\ \texttt{else}\ g' \ = \ \texttt{if}\ [x := s]\ e\ \texttt{then}\ [x := s]\ g\ \texttt{else}\ [x := s]\ g'$$

Figure 21. Substitution relation.

Inversion Lemmas

```
Lemma Inversion-Subtype-Arrow:
    forall K S'' T T',
        K |- S'' <: (T -> T') -->
        (exists S S',
            S'' = (S -> S') /\
            K |- T <: S /\ K |- S' <: T').

Proof by inversion on subtyping:

CASE: Sub-Refl
    S'' = (T -> T')
    K |- S'' <: (T -> T')
    ---------------------
    exists T T'
    by Sub-Refl, we show K |- T <: T and K |- T' <: T'.

CASE: Sub-Fun
    K |- T <: S
    K |- S' <: T'
    S'' = S -> S'
    K |- S'' <: (T -> T')
    ----------------------
    exists S S',
    by hypotheses we show K |- T <: S and K |- S' <: T'

CASE: Sub-Fam
    K |- a ~> L
    R = {(f_d : T_d)*} in L.TYPES
    K |- {(f_d : T_d)*} <: (T -> T') % contradiction
    S'' = a.R
    K |- S'' <: (T -> T')
    ---------------------

    K |- {(f_i: T_i)*} <: (T -> T') - contradiction,
    there is no subtyping rule that allows this


=============================================================================

Lemma Inversion-Subtype-Record:
    forall S f_j* T_j*,
        K |- S <: {(f_j: T_j)*} -->
        (exists f_i* T_i*,
            S = {(f_i: T_i)*} /\
            forall j, exists T, K |- T <: T_j /\ (f_j : T ) in (f_i: T_i)*)
        \/ (exists a R f_d* T_d* L,
            S = a.R /\ K |- a ~> L /\ R = {(f_d: T_d)*} in L.TYPES /\ K |- {(f_d: T_d)*} <: {(f_j: T_j)*})

Proof by inversion on subtyping:

CASE: Sub-Refl
    S = {(f_j: T_j)*}
    K |- S <: {(f_j: T_j)*}
    ----------------
    exists f_j* T_j*, S = {(f_j: T_j)*} by hypothesis
    /\ K |- {(f_j: T_j)*} <: {(f_j: T_j)*}) by Sub-Refl


CASE: Sub-Fam
    K |- a ~> L
    R = {(f_d: T_d)*} in L.TYPES
    K |- {(f_d: T_d)*} <: {(f_j: T_j)*}
    S = a.R
    K |- S <: {(f_j: T_j)*}
    -------------------
    exists a R f_d* T_d* L,
        S = a.R /\ K |- a ~> L /\ R = {(f_d: T_d)*} in L.TYPES /\ K |- {(f_d: T_d)*} <: {(f_j: T_j)*}
        by hypotheses

CASE: Sub-Rec
    forall j, exists T, K |- T <: T_j /\ (f_j : T ) in (f_i: T_i)*
    S = {(f_i: T_i)*}
    K |- S <: {(f_j: T_j)*}
    -------------------------------------
    exists f_i* T_i*,
        S = {(f_i: T_i)*} /\
        forall j, exists T, K |- T <: T_j /\ (f_j : T ) in (f_i: T_i)*)
        by hypotheses

=============================================================================
```

Canonical Forms

```
Lemma Canonical-Fun:
    forall K G v T T',
        K, G |- v : (T -> T') -->
        exists x S e, K |- T <: S /\ v == (lam (x: S). e)

Proof by induction on the typing derivation:

CASES: T-Var, T-App, T-Proj, T-FamFun, T-Cases, T-Match, T-If
    ------------------
    contradiction: expression not a value

CASE: T-Lam
    v = lam (x : T). e
    K, G |- v : T -> T'
    --------------------------------
    exists x T e, K |- T <: T by sub-refl
    and v == (lam (x: T). e) by hypothesis


CASE: T-Subsumption

    K, G |- v : Tsub
    K |- Tsub <: (T -> T')
    K, G |- v : (T -> T')
    -------------------
    exists x S e, T <: S /\ v == (lam (x: S). e)

    By Inversion-Subtype-Arrow lemma, every subtype of an arrow type is an arrow type
    so: Tsub = (Tsub' -> Tsub'')
    Then, we know that K, G |- v : (Tsub' -> Tsub'')

    use induction on K, G |- v : Tsub with Tsub = (Tsub' -> Tsub'')

    by induction hypothesis:
        K, G |- v : (Tsub' -> Tsub'') -->
        exists x S e, K |- Tsub' <: S /\ v == (lam (x: S). e)

    we know from K |- Tsub <: (T -> T') that K |- T <: Tsub'
    if K |- T <: Tsub', and K |- Tsub' <: S, then by transitivity of subtyping K |- T <: S

    Therefore, exists x S e, K |- T <: S /\ v == (lam (x: S). e).

=============================================================================


Lemma Canonical-Rec:
    forall K G v f_i* T_i*,
        K, G |- v : {(f_i: T_i)*} -->
        (exists f_j* v_j*, v == {(f_j = v_j)*}
            /\ forall i,
            (exists v_i, (f_i = v_i) in (f_j = v_j)* /\ K, G |- v_i : T_i))
        \/ (exists f_j* v_j* a R, v == a.R({(f_j = v_j)*}) /\ K, G |- a.R({(f_j = v_j)*}) : a.R
            /\ K |- a.R <: {(f_i: T_i)*})

Proof by induction on the typing derivation:
(premises from the typing judgment included as needed labeled H1, H2, etc)

CASES: T-Var, T-App, T-Proj, T-Cases, T-Match, T-If
    --------------------------------
    contradiction: expression not a value

CASE: T-Rec
    H1: forall i, K, G |- e_i : T_i
    v = {(f_i = e_i)*}
    K, G |- v : {(f_i: T_i)*}
    -------------------------
    Since v is a value, we know that all e_i's are values (v_i)
    Thus, we know that v is some record of values {(f_i = v_i)*}
        (which satisfies part 1 of goal: exists f_j* v_j*, v == {(f_j = v_j)*}).
    We also know that since
```

```
        forall i, G |- v_i : T_i,
    for each field f_i there exists a value with type T_i in the definition of v
        (satisfies part 2 of goal:
            forall i, (exists v_i, (f_i = v_i) in (f_j = v_j)* /\ K, G |- v_i : T_i))


CASE: T-Subs
    H1: K, G |- v : S
    H2: K |- S <: {(f_i: T_i)*}
    K, G |- v : {(f_i: T_i)*}
    ------------------

    By Inversion-Subtype-Record lemma, S can be a record type or a named family type.

    Case 1: S is some record type.
        Then, exists f_s* T_s*, S = {(f_s: T_s)*} /\
                forall i, exists T, K |- T <: T_i /\ (f_i : T) in (f_s: T_s)*

    Rewrite state:

        H1: K, G |- v : {(f_s: T_s)*}
        H2: K |- {(f_s: T_s)*} <: {(f_i: T_i)*}
        forall i, exists T, K |- T <: T_i /\ (f_i : T) in (f_s: T_s)*
        K, G |- v : {(f_i: T_i)*}
        ------------------

        By induction hypothesis:
            K, G |- v : {(f_s: T_s)*} -->
            (exists f_j* v_j*, v == {(f_j = v_j)*}
                /\ forall s, (exists v_s, (f_s = v_s) in (f_j = v_j)* /\ K, G |- v_s : T_s))
            \/ (exists f_j* v_j* a R, v == a.R({(f_j = v_j)*}) /\ K, G |- a.R({(f_j = v_j)*}) : a.R
                /\ K |- a.R <: {(f_s: T_s)*})
        (v is either a record value or an instance value)

        Subcase 1: v is a record value
            Then the following hold:
                exists f_j* v_j*, v == {(f_j = v_j)*}
                /\ forall s, (exists v_s, (f_s = v_s) in (f_j = v_j)* /\ K, G |- v_s : T_s)
            Need to show:
                exists f_j* v_j*, v == {(f_j = v_j)*}
                /\ forall i, (exists v_i, (f_i = v_i) in (f_j = v_j)* /\ K, G |- v_i : T_i)
            First subgoal (exists f_j* v_j*, v == {(f_j = v_j)*}) is already shown in proper form.
            For the second subgoal:
            + We know that for each field f_s there is a value v_s
                with some type T_s in definition of v.
            + We also know that each field in {(f_i: T_i)*} appears in {(f_s: T_s)*}
                with some type T that is a subtype of T_i by premise:
                forall i, exists T, K |- T <: T_i /\ (f_i : T) in (f_s: T_s)*
            + Hence, from
                forall s, (exists v_s, (f_s = v_s) in (f_j = v_j)* /\ K, G |- v_s : T_s)
                we can derive
                forall i, (exists v_i, (f_i = v_i) in (f_j = v_j)* /\ K, G |- v_i : T_i)
                since all fields f_i appear in the set of fields f_s*
                and for each value v_s,
                since we have K, G |- v_s : T_s and K |- T_s <: T_i,
                we can derive K, G |- v_s : T_i by T-Subsumption.


        Subcase2: v is an instance value
            Then the following hold:
                (exists f_j* v_j* a R, v == a.R({(f_j = v_j)*}) /\ K, G |- a.R({(f_j = v_j)*}) : a.R
                    /\ K |- a.R <: {(f_s: T_s)*})
            Need to show:
                (exists f_j* v_j* a R, v == a.R({(f_j = v_j)*}) /\ K, G |- a.R({(f_j = v_j)*}) : a.R
                    /\ K |- a.R <: {(f_i: T_i)*})
            The top line of the goal matches what we already have verbatim.
            Since we have K |- a.R <: {(f_s: T_s)*} and by premise H2: K |- {(f_s: T_s)*} <: {(f_i: T_i)*},
            we derive K |- a.R <: {(f_i: T_i)*}.


    Case2: S is a named family type.
    Then we have
    (exists a R f_d* T_d* L,
             S = a.R /\ K |- a ~> L /\ R = {(f_d: T_d)*} in L.TYPES /\ K |- {(f_d: T_d)*} <: {(f_i: T_i)*})

    Rewrite state:
        H1: K, G |- v : a.R
        H2: K |- a.R <: {(f_i: T_i)*}
        K |- a ~> L
        R = {(f_d: T_d)*} in L.TYPES
        K |- {(f_d: T_d)*} <: {(f_i: T_i)*}
        K, G |- v : {(f_i: T_i)*}
        ------------------

        Apply Canonical-Fam to H1.
        By Canonical-Fam, we know that v is either an instance of a named record type a.R
        or an instance of an ADT a.R.

        case1: v is an instance of an ADT a.R
            Then, the following hold by Canonical-Fam:
            exists C f_k* v_k* L' C_j f_i'* T_i'* T_k*,
            v == a.R(C {(f_k = v_k)*}) /\ K |- a ~> L' /\
            R = \overline{C_j {(f_i' : T_i')*}} in L'.ADTS /\
            C {(f_k : T_k)*} in \overline{C_j {(f_i' : T_i')*}} /\
            forall i', K, G |- v_k : T_k

            L and L' refer to the same linkage because they are produced from the
            same path a. Hence, we have two conflicting definitions for R:

            R = {(f_d: T_d)*} in L.TYPES and
            R = \overline{C_j {(f_i' : T_i')*}} in L.ADTS

            We cannot have types of the same name with different definitions
            by well-formedness of linkage L, so this is a contradiction.

        case2: v is an instance of a named record type a.R
            Then, the following hold by Canonical-Fam:

            exists f_i'* v_i'* T_i'* L',
            v == a.R({(f_i' = v_i')*}) /\ K |- a ~> L' /\
            R = {(f_i' : T_i')*} in L'.TYPES /\ forall i, K, G |- v_i' : T_i'

            need to show:
            (exists f_j* v_j* a' R', v == a'.R'({(f_j = v_j)*}) /\ K, G |- a'.R'({(f_j = v_j)*}) : a'.R'
            /\ K |- a'.R' <: {(f_i: T_i)*})

            Take f_j* = f_i'*, v_j* = v_i'*, a' = a, R' = R, rewrite goal:
            v == a.R({(f_i'= v_i')*}) /\ K, G |- a.R({(f_i' = v_i')*}) : a.R
            /\ K |- a.R <: {(f_i: T_i)*}

            Leftmost subgoal and rightmost subgoal are already shown.
            We derive K, G |- a.R({(f_i' = v_i')*}) : a.R from
            K |- a ~> L' /\ R = {(f_i' : T_i')*} in L'.TYPES /\ forall i, K, G |- v_i' : T_i'
            by T_Constr.

================================================================================

Lemma Canonical-Fam:
    forall K G v a R,
        K, G |- v : a.R -->
        (exists f_i* v_i* T_i* L,
            v == a.R({(f_i = v_i)*}) /\ K |- a ~> L /\
            R = {(f_i : T_i)*} in L.TYPES /\ forall i, K, G |- v_i : T_i)
            \/
        (exists C f_k* v_k* L C_j f_i* T_i* T_k*,
            v == a.R(C {(f_k = v_k)*}) /\ K |- a ~> L /\
            R = \overline{C_j {(f_i : T_i)*}} in L.ADTS /\
            C {(f_k : T_k)*} in \overline{C_j {(f_i : T_i)*}} /\
            forall i, K, G |- v_k : T_k)

Proof by induction on the typing derivation:

CASES: T-Var, T-App, T-Proj, T-Cases, T-Match, T-If
    --------------------------------
    contradiction: expression not a value

CASE T-Constr:
    by induction on the typing derivation, we have:
        v == a.R({(f_i = e_i)*}) /\
            K |- a ~> L /\ R = {(f_i : T_i)*} in L.TYPES /\ forall i, K, G |- e_i : T_i.
    Since v is a value, the e_i's are also values v_i.
    Thus, (exists f_i* v_i* T_i* L,
            v == a.R({(f_i = v_i)*}) /\ K |- a ~> L /\
            R = {(f_i : T_i)*} in L.TYPES /\ forall i, K, G |- v_i : T_i)

T-ADT:
    by induction on the typing derivation, we have:
        v == a.R(C {(f_k = e_k)*}) /\
            K |- a ~> L /\ R = \overline{C_j {(f_i : T_i)*}} in L.ADTS /\
            C {(f_k: T_k)*} in \overline{C_j {(f_i : T_i)*}} /\
            forall k, K, G |- e_k : T_k
    Since v is a value, the e_i's are also values v_i
    Thus, (exists C f_k* v_k* L C_j f_i* T_i* T_k*,
            v == a.R(C {(f_k = v_k)*}) /\ K |- a ~> L /\
            R = \overline{C_j {(f_i : T_i)*}} in L.ADTS /\
            C {(f_k : T_k)*} in \overline{C_j {(f_i : T_i)*}} /\
            forall i, K, G |- v_k : T_k)

T-Subs:
        by induction on the typing derivation, we have
        K, G |- v : T'
        K |- T' <: a.R

        The only subtype of a.R is a.R (meaning T' = a.R),
        so we use the induction hypothesis
        on K, G |- v : T' to prove the goal.

=============================================================================
```

Transitivity of Subtyping

```
Lemma Sub-Trans:
    forall K T1 T2 T3,
        K |- T1 <: T2 -->
        K |- T2 <: T3 -->
        K |- T1 <: T3.

Proof by induction on T2.

CASE 1: T2 = N
    N does not have subtypes. Contradiction in premise 1.


CASE 2: T2 = B
    B does not have subtypes. Contradiction in premise 1.


CASE 3: T2 = a.R

    K |- T1 <: a.R
    K |- a.R <: T3
    -----------------

    The only subtyping rule that applies to premise 1 is Sub-Refl.
    Thus, T1 = a.R and we already know a.R <: T3 by premise 2.


CASE 4: T2 = T2in -> T2out

    K |- T1 <: (T2in -> T2out)
    K |- (T2in -> T2out) <: T3
    ------------------------

    Subtyping rules that can apply to premise 1: Sub-Refl, Sub-Fun

    Subcase 1: in premise 1, Sub-Refl applies.
        Then, T1 = T2in -> T2out, and premise 2 finishes the proof.

    Subcase 2: in premise 1, Sub-Fun applies.
        Then, the following is also true:
        T1 = (T1in -> T1out)
        K |- T2in <: T1in
        K |- T1out <: T2out.

        Subtyping rules that can apply to premise 2: Sub-Refl, Sub-Fun
        If Sub-Refl applies, solution is trivial by premise 1.

        If Sub-Fun applies, we also know the following:
        T3 = (T3in -> T3out)
        K |- T3in <: T2in
        K |- T2out <: T3out.

        We need to show:
        (T1in -> T1out) <: (T3in -> T3out)
        or, equivalently:
        (K |- T3in <: T1in) and (K |- T1out <: T3out).

        By induction hypothesis, we know that:
        - since K |- T3in <: T2in, and K |- T2in <: T1in, then K |- T3in <: T1in
        - since K |- T1out <: T2out, and K |- T2out <: T3out, then K |- T1out <: T3out

        Proven. No other subcases.


CASE 5: T2 = {(f_j: T_j)*}

    K |- T1 <: {(f_j: T_j)*}
    K |- {(f_j: T_j)*} <: T3
    --------------------------
```

```
Subtyping rules that can apply to premise 1: Sub-Refl, Sub-Rec, Sub-Fam

Subcase 1: In premise 1, Sub-Refl applies.
    Then, T1 = {(f_j: T_j)*} and proven by premise 2.

Subcase 2: In premise 1, Sub-Rec applies.
    Then, we know the following:
    T1 = {(f_i: T_i)*} AND
    forall j,
        (exists T, K |- T <: T_j -->
        (f_j: T) in {(f_i: T_i)*})

    Subtyping rules that can apply to premise 2: Sub-Refl, Sub-Rec
    If Sub-Refl applies, solution is trivial by premise 1.

    If Sub-Rec applies, we also know the following:
    T3 = {(f_k: T_k)*} AND
    forall k,
        (exists T', K |- T' <: T_k -->
        (f_k: T') in {(f_j: T_j)*})

    We need to show that:
    T1 <: T3, or equivalently that:
    forall k,
        (exists Ts, K |- Ts <: T_k -->
        (f_k: Ts) in {(f_i: T_i)*})

    For each field in T3, we have the same field with a subtype in T2,
    and for each field in T2, we have the same field with a subtype in T1.
    Thus, each field in T3 that appears in T1 has a type in T1 that is a subtype
    of the field in T3 (true by induction hypothesis).

    More specifically, for each field of T3 that appears in T1, Ts == T.

Subcase 3: In premise 1, Sub-Fam applies.
    Then, we have T1 = a.R, and:

    K |- a ~> L
    R = {(f_d: T_d)*} in L.TYPES
    K |- {(f_d: T_d)*} <: {(f_j: T_j)*}

    Now, we have to show transitivity for the case where T1 and T2 are
    record types. The proof is identical to Subcase 2.
```

Substitution is type-preserving

```
Assume s does not have free variables.

Statement:

K, (x: X, G) |- e : T -->
K, G |- s : X -->
K, G |- [x:=s] e : T

Proof by induction on the typing derivation of e.
State includes hypotheses from inversion of the typing derivation where needed (H1, H2, etc)

CASE: T-Num

    K, (x: X, G) |- n : N
    K, G |- s : X
    -------------
    K, G |- [x:=s] n : N

    + By S_Nat: [x:=s] n = n
    + Rewrite goal:
        K, G |- n : N
    + Since n does not have any free variables, x in the typing context is irrelevant.
    + Thus, from premise we derive K, G |- n : N


CASE: T-Bool

    K, (x: X, G) |- b : B
    K, G |- s : X
    -------------
    K, G |- [x:=s] b : B

    + By S_Bool: [x:=s] b = b
    + Rewrite goal:
        K, G |- b : B
    + Since b does not have any free variables, x in the typing context is irrelevant.
    + Thus, from premise we derive K, G |- b : B


CASE: T-Var

    H1: y: T \in (x: X, G)
    K, (x: X, G) |- y : T
    K, G |- s : X
    -------------
    K, G |- [x:=s] y : T

    Case 1: y != x
        + By S_VarNeq: [x:=s] y = y
        + Rewrite goal:
            K, G |- y : T
        + Since we have y: T \in (x: X, G) and also y != x, it must
        be true that y: T \in G
        + By T-Var, because y: T \in G we derive K, G |- y : T.


    Case 2: y == x
        + Then, we also know T == X (from H1)
        + Rewrite state:

            K, (y: T, G) |- y : T
            K, G |- s : T
            -------------
            K, G |- [y:=s] y : T

        + By S_Var: [x:=s] x = s
        + Rewrite goal: K, G |- s : T
        + True by hypothesis
```

```
CASE: T-Lam

    H1: K |- WF(T)
    H2: K, (y : T, x: X, G) |- e : T'
    K, (x: X, G) |- lam (y : T). e : T -> T'
    K, G |- s : X
    -----------------------------------
    K, G |- [x:=s] lam (y : T). e : T -> T'

    Case 1: y == x
    + By S_Lam: [x:=s] lam (x: T). e = lam (x: T). e
    + Rewrite goal:
        K, G |- lam (x : T). e : T -> T'
    + since we know that
        K, (x: X, G) |- lam (x : T). e : T -> T'
      and x does not occur free in e, the x in the context is irrelevant to typing of e.
      Thus we can derive: K, G |- lam (x : T). e : T -> T'


    Case 2: y != x
    + By S-LamNeq: [x:=s] lam (y: T). e = lam (y: T). ([x:=s] e)
    + Rewrite goal:
        K, G |- lam (y: T). ([x:=s] e) : T -> T'
    + By induction hypothesis, we have
        K, (y : T, x: X, G) |- e : T' -->
        K, (y: T, G) |- s : X -->
        K, (y: T, G) |- [x:=s] e : T'
    + Since we have K, G |- s : X, we can show K, (y: T, G) |- s : X as long as s does not
        have a free variable y. Since s does not have free variables, this is true.
    + Since we have K |- WF(T), and K, (y: T, G) |- [x:=s] e : T' from the induction hypothesis,
        we derive K, G |- lam (y: T). ([x:=s] e) : T -> T' by T-Lam.


CASE: T-App

    H1: K, (x: X, G) |- e : T
    H2: K, (x: X, G) |- g : T -> T'
    K, (x: X, G) |- g e : T'
    K, G |- s : X
    --------------------
    K, G |- [x:=s] g e : T'

    + By S_App: [x:=s] g e = ([x:=s] g) ([x:=s] e)
    + Rewrite goal:
        K, G |- ([x:=s] g) ([x:=s] e) : T'

    + By induction hypotheses, we have
        K, (x: X, G) |- e : T -->
        K, G |- s : X -->
        K, G |- [x:=s] e : T
        AND
        K, (x: X, G) |- g : T -> T' -->
        K, G |- s : X -->
        K, G |- [x:=s] g : T -> T'

    + Since we have
        K, G |- [x:=s] e : T
        and K, G |- [x:=s] g : T -> T',
        we derive K, G |- ([x:=s] g) ([x:=s] e) : T' by T-App.


CASE: T-Rec

    forall i, Hi:
        K, (x: X, G) |- e_i : T_i
    K, (x: X, G) |- {(f_i = e_i)*} : {(f_i: T_i)*}
    K, G |- s : X
    -------------------------------------
    K, G |- [x:=s] {(f_i = e_i)*} : {(f_i: T_i)*}

    + By S_Rec: [x:=s] {(f_i = e_i)*} = {(f_i = ([x:=s] e_i))*}
    + Rewrite goal:
        K, G |- {(f_i = ([x:=s] e_i))*} : {(f_i: T_i)*}
    + By induction hypothesis INDi for each i:
        K, (x: X, G) |- e_i : T_i -->
        K, G |- s : X -->
        K, G |- ([x:=s] e_i) : T_i
    + Since by INDi we have:
        forall i,
            K, G |- ([x:=s] e_i) : T_i
        we derive K, G |- {(f_i = ([x:=s] e_i))*} : {(f_i: T_i)*} by T-Rec.


CASE: T-Proj

    H1: K, (x: X, G) |- e : {(f_i: T_i)*}
    H2: (f: T) in (f_i: T_i)*
    K, (x: X, G) |- e.f : T
    K, G |- s : X
    -------------------------------------
    K, G |- [x:=s] e.f : T

    + By S_Proj: [x:=s] e.f = ([x:=s] e).f
    + Rewrite goal:
        K, G |- ([x:=s] e).f : T
    + By induction hypothesis, we have
        K, (x: X, G) |- e : {(f_i: T_i)*} -->
        K, G |- s : X -->
        K, G |- [x:=s] e : {(f_i: T_i)*}
    + Since we have
        K, G |- [x:=s] e : {(f_i: T_i)*}
        and (f: T) in (f_i: T_i)*,
        we derive K, G |- ([x:=s] e).f : T by T-Proj.


CASE: T-FamFun

    K, (x: X, G) |- a.m : T -> T'
    K, G |- s: X
    -----------------------------
    K, G |- [x:=s] a.m : T -> T'

    + By S_FamFun: [x:=s] a.m = a.m
    + Rewrite goal:
        K, G |- a.m : T -> T'
    + We have K, (x: X, G) |- a.m : T -> T'.
    Since a.m does not have any free variables, x cannot appear in a.m,
    and is irrelevant as part of the typing context.
    Thus, we derive K, G |- a.m : T -> T'


CASE: T-Cases

    K, (x: X, G) |- a.c : {(f_i:T_i)*} -> {(C_j:T_j->T_j')*}
    K, G |- s: X
    -----------------------------
    K, G |- [x:=s] a.c : {(f_i:T_i)*} -> {(C_j:T_j->T_j')*}

    + By S_Cases: [x:=s] a.c = a.c
    + Rewrite goal:
        K, G |- a.c : {(f_i:T_i)*} -> {(C_j:T_j->T_j')*}
    + We have K, (x: X, G) |- a.c : {(f_i:T_i)*} -> {(C_j:T_j->T_j')*}.
    Since a.c does not have any free variables, x cannot appear in a.c,
    and is irrelevant as part of the typing context.
    Thus, we derive G |- a.c : {(f_i:T_i)*} -> {(C_j:T_j->T_j')*}


CASE: T-Constr

    H1: K |- a ~> L
    H2: R = {(f_i: T_i)*} in L.TYPES
    forall i, Hi:
        K, (x: X, G) |- e_i : T_i
    K, (x: X, G) |- a.R({(f_i = e_i)*}) : a.R
    K, G |- s : X
    -----------------------------------
    K, G |- [x:=s] a.R({(f_i = e_i)*}) : a.R

    + By S_Constr: [x:=s] a.R({(f_i = e_i)*}) = a.R({(f_i = ([x:=s] e_i))*})
    + Rewrite goal:
        K, G |- a.R({(f_i = ([x:=s] e_i))*}) : a.R
    + By induction hypothesis INDi for each i:
        K, (x: X, G) |- e_i : T_i -->
        K, G |- s : X -->
        K, G |- ([x:=s] e_i) : T_i
    + Since by IND we have:
        forall i, K, G |- ([x:=s] e_i) : T_i
        and from premises H1 and H2,
        we derive K, G |- a.R({(f_i = ([x:=s] e_i))*}) : a.R by T-Constr.



CASE: T-ADT

    H1: K |- a ~> L
    H2: R = \overline{C_j {(f_i: T_i)*}} in L.ADTS
    H3: C {(f_k: T_k)*} in \overline{C_j {(f_i: T_i)*}}
    forall k, Hk:
        K, (x: X, G) |- e_k : T_k
    K, (x: X, G) |- a.R(C {(f_k = e_k)*}) : a.R
    K, G |- s : X
    -------------------------------------------
    K, G |- [x:=s] a.R(C {(f_k = e_k)*}) : a.R

    + By S_ADT: [x:=s] a.R(C {(f_i = e_i)*}) = a.R(C {(f_i = ([x:=s] e_i))*})
    + Rewrite goal:
        K, G |- a.R(C {(f_k = ([x:=s] e_k))*}) : a.R
    + By induction hypothesis INDi for each i:
        K, (x: X, G) |- e_k : T_k -->
        K, G |- s : X -->
        K, G |- ([x:=s] e_k) : T_k
    + Since by IND we have:
        forall k,
            K, G |- ([x:=s] e_k) : T_k
        and from premises H1, H2, and H3,
        we derive K, G |- a.R(C {(f_k = ([x:=s] e_k))*}) : a.R by T-ADT.



CASE: T-Match

    H1: K |- a' ~> L
    H2: K, (x: X, G) |- e : a'.R
    H3: R = \overline{C_j {(f_i: T_i)*}} in L.ADTS
    H4: K, (x: X, G) |- a.c : {(f_arg: T_arg)*} -> {(C_j: {(f_i: T_i)*} -> T)*}
    H5: K, (x: X, G) |- {(f_arg = e_arg)*} : {(f_arg: T_arg)*}
    K, (x: X, G) |- match e with a.c {(f_arg = e_arg)*} : T
    K, G |- s : X
    ---------------------------------------------------------
    K, G |- [x:=s] match e with a.c {(f_arg = e_arg)*} : T

    + By S_Match:
        [x:=s] match e with a.c {(f_arg = e_arg)*} =
          match ([x:=s] e) with a.c ([x:=s] {(f_arg = e_arg)*})
    + Rewrite goal:
        K, G |- match ([x:=s] e) with a.c ([x:=s] {(f_arg = e_arg)*}) : T
    + By induction hypothesis on e, we have:
        K, (x: X, G) |- e : a'.R -->
        K, G |- s : X -->
        K, G |- [x:=s] e : a'.R
    + By induction hypothesis on {(f_arg = e_arg)*}, we have:
        K, (x: X, G) |- {(f_arg = e_arg)*} : {(f_arg: T_arg)*} -->
        K, G |- s : X -->
        K, (x: X, G) |- [x:=s] {(f_arg = e_arg)*} : {(f_arg: T_arg)*}
    + Since we have
        K, G |- [x:=s] e : a'.R and
        K, (x: X, G) |- [x:=s] {(f_arg = e_arg)*} : {(f_arg: T_arg)*}
        by induction and also have H1, H3, H4,
        we can derive
        K, G |- match ([x:=s] e) with a.c ([x:=s] {(f_arg = e_arg)*}) : T
        by T-Match.


CASE: T-Subs

    H1: K, (x: X, G) |- e : T'
    H2: K |- T' <: T
    K, (x: X, G) |- e : T
    K, G |- s : X
    --------------------
    K, G |- [x:=s] e : T

    By induction hypothesis:
    K, (x: X, G) |- e : T' -->
    K, G |- s : X -->
    K, G |- [x:=s] e : T'

    Since we know K, G |- [x:=s] e : T' and K |- T' <: T, we can
    derive K, G |- [x:=s] e : T by rule T_Subs.


CASE: T-If

    H1: K, (x: X, G) |- e : B
    H2: K, (x: X, G) |- g : T
    H3: K, (x: X, G) |- g' : T
    K, (x: X, G) |- if e then g else g': T
    K, G |- s : X
    -----------------------------------------
    K, G |- [x:=s] if e then g else g': T

    + by S_If: [x:=s] if e then g else g' =
        if ([x:=s] e) then ([x:=s] g) else ([x:=s] g')
    + Rewrite goal:
        K, G |- if ([x:=s] e) then ([x:=s] g) else ([x:=s] g') : T
    + by induction, we have:
        K, (x: X, G) |- e : B -->
        K, G |- s : X -->
        K, G |- [x:=s] e : B
    AND
        K, (x: X, G) |- g : T -->
        K, G |- s : X -->
        K, G |- [x:=s] g : T
    AND
        K, (x: X, G) |- g' : T -->
        K, G |- s : X -->
        K, G |- [x:=s] g' : T
    + Since we have
        K, G |- [x:=s] e : B,
        K, G |- [x:=s] g : T, and
        K, G |- [x:=s] g' : T,
        we derive
        K, G |- if ([x:=s] e) then ([x:=s] g) else ([x:=s] g') : T
        by T-If.
```

Progress Nested

```
Statement:
    For any expression e in program p,
    []; [] |- p : T_p /\
    [prog]; [] |- e : T' -->
    value(e) \/ exists e', [prog] |- e ---> e'.

Proof by induction on the typing derivation [prog]; [] |- e : T'.

The following are already included in the premise when necessary:
    - premises from induction on the typing derivation (named H1, H2, etc)
    - induction hypotheses (named IND1, IND2, etc)

CASE: T-Num

    []; [] |- p : T_p
    [prog]; [] |- n: N
    ------------------
    value(n)


CASE: T-Bool

    []; [] |- p : T_p
    [prog]; [] |- b: B
    ------------
    value(b)


CASE: T-Var

    []; [] |- p : T_p
    [prog]; [] |- x : T'
    H1: x:T' \in []
    -------------
    Contradiction in H1


CASE: T-Lam

    []; [] |- p : T_p
    [prog]; [] |- lam (x : T'). e : T' -> T''
    ----------------------------------
    value (lam (x : T'). e)


CASE: T-App

    []; [] |- p : T_p
    [prog]; [] |- g e : T'
    H1: [prog]; [] |- e : T
    H2: [prog]; [] |- g : T -> T'
    IND1: value(e) \/ exists e', [prog] |- e ---> e'
    IND2: value(g) \/ exists g', [prog] |- g ---> g'
    --------------------

    case1: g can be reduced
        + exists g', [prog] |- g ---> g'
        + thus,
            exists g' e, [prog] |- g e ---> g' e
            by R-App

    case2: value(g); exists e', [prog] |- e ---> e'
        + thus,
            exists v e', [prog] |- v e ---> v e'
            by R-LamArg

    case3: value(g), value(e)
        + g is a value of type T -> T', and thus by lemma Canonical-Fun
        it can only be of form (lam (x: S). e') where [prog] |- T <: S.
        + thus, since g = (lam (x: S). e') and e is a value (v),
            exists ([x:=v] e'), [prog] |- (lam (x: S). e') v ---> [x:=v] e'
            by R-LamApply.


CASE: T-Rec

    []; [] |- p : T_p
    [prog]; [] |- {(f_i = e_i)*} : {(f_i: T_i)*}
    forall i,
        Hi: [prog]; [] |- e_i : T_i
    forall i,
        INDi: value(e_i) \/ exists e_i', [prog] |- e_i ---> e_i'
    -------------------------------------

    By IND_i,
    each e_i is either a value or can reduce.

    If all e_i's are values v_i, then the record {(f_i = v_i)*} is a value.

    If for some i, e_i is the first non-value, then the record can reduce.

    consider e_i in order from left to right
    all expressions in the record at indices before e_i are values

    case1: e_i can reduce
        + use induction hypothesis for e_i:
            exists e', [prog] |- e_i ---> e_i'
        + thus,
            exists {f_0 = v_0, ..., f_i = e_i', ...},
            [prog] |- {f_0 = v_0, ..., f_i = e_i, ...} --->
            {f_0 = v_0, ..., f_i = e_i', ...}
            by R-Rec

    case2:
        + when we're done reducing everything:
            value ({(f_i = v_i)*})


CASE: T-FamFun

    []; [] |- p : T_p
    [prog]; [] |- a.m : T -> T'
    H1: [prog] |- a ~> L
    H2: m : T -> T' = lam(x:T).e in L.FUNS
    ---------------------------------------------

    + exists lam(x:T).e,
        [prog] |- a.m ---> lam(x:T).e
        by R-FamFun


CASE: T-Cases

    []; [] |- p : T_p
    [] |- a.c : {(f_i:T_i)*} -> {(C_j: T_j->T_j')*}
    H1: [prog] |- a ~> L
    H2: c <a'.R> : {(f_i:T_i)*} -> {(C_j: T_j->T_j')*} =
        lam (x: {(f_i:T_i)*}). {(C_j = lam (y_j: T_j). e_j)*} in L.CASES
    ---------------------------------------------

    + exists lam (x: {(f_i:T_i)*}). {(C_j = lam (y_j: T_j). e_j)*},
        [prog] |- a.c ---> lam (x: {(f_i:T_i)*}). {(C_j = lam (y_j: T_j). e_j)*}
        by R-Cases


CASE: T-Constr

    []; [] |- p : T_p
    [prog]; [] |- a.R({(f_i = e_i)*}) : a.R
    H1: [prog] |- a ~> L
    H2: R = {(f_i: T_i)*} in L.TYPES
    forall i,
        Hi: [prog]; [] |- e_i : T_i
    forall i,
        INDi: value(e_i) \/ exists e_i', [prog] |- e_i ---> e_i'
    -----------------------------------

    case 1: there is an e_i that can reduce
        + Take the first e_i that can reduce, then our record is of this form:
            {f_0 = v_0, ..., f_(i-1) = v_(i-1), f_i = e_i, ...}
        + Since [prog] |- e_i ---> e_i', we have
            [prog] |- {f_0 = v_0, ..., f_(i-1) = v_(i-1), f_i = e_i, ...} --->
            {f_0 = v_0, ..., f_(i-1) = v_(i-1), f_i = e_i', ...}
            by R-Rec
        + thus,
            exists {f_0 = v_0, ..., f_(i-1) = v_(i-1), f_i = e_i', ...},
            [prog] |- a.R({f_0 = v_0, ..., f_(i-1) = v_(i-1), f_i = e_i, ...}) --->
            a.R({f_0 = v_0, ..., f_(i-1) = v_(i-1), f_i = e_i', ...})
            by R-Instance

    case2: all e_i are values
        + we have a.R({(f_i = v_i)*}) which is a value


CASE: T-ADT

    []; [] |- p : T_p
    [prog]; [] |- a.R(C {(f_k = e_k)*}) : a.R
    H1: [prog] |- a ~> L
    H2: R = \overline{C_j {(f_i: T_i)*}} in L.ADTS
    H3: C {(f_k: T_k)*} in \overline{C_j {(f_i: T_i)*}}
    forall k,
       Hk: [prog]; [] |- e_k : T_k
    forall k,
        INDk: value(e_k) \/ exists e_k', [prog] |- e_k ---> e_k'
    ------------------------------------------
    case 1: there is an e_k that can reduce
        + Take the first e_k that can reduce, then our record is of this form:
            {f_0 = v_0, ..., f_(k-1) = v_(k-1), f_k = e_k; ...}
        + Since [prog] |- e_k ---> e_k’, we have
            [prog] |- {f_0 = v_0, ..., f_(k-1) = v_(k-1), f_k = e_k; ...} --->
            {f_0 = v_0, ..., f_(k-1) = v_(k-1), f_k = e_k’, ...}
            by R-Rec
        + thus,
            exists {f_0 = v_0, ..., f_(k-1) = v_(k-1), f_k = e_k’, ...},
            [prog] |- a.R(C {f_0 = v_0, ..., f_(k-1) = v_(k-1), f_k = e_k; ...}) --->
            a.R(C {f_0 = v_0, ..., f_(k-1) = v_(k-1), f_k = e_k’, ...})
            by R-ADT

    case2: all e_k are values
        + we have a.R(C {(f_k = v_k)*}) which is a value


CASE: T-Subs

    []; [] |- p : T_p
    [prog]; [] |- e : T’
    H1: [prog]; [] |- e : T
    H2: [prog] |- T <: T’
    IND1: value(e) \/ exists e’, [prog] |- e ---> e’
    --------------------
    IND


CASE: T-Proj

    []; [] |- p : T_p
    [prog]; [] |- e.f : T’
    H1: [prog]; [] |- e : {(f_i: T_i)*}
    H2: (f: T) in (f_i: T_i)*
    IND: value(e) \/ exists e’, [prog] |- e ---> e’
    -------------------------------------

    case1: exists e’, [prog] |- e ---> e’
        + thus,
            exists e’.f, [prog] |- e.f ---> e’.f
            by R-Proj

    case2: value(e)
        by Canonical-Rec, since e has a record type, there are 2 cases:

        case 2.1: e is a record expression
        + Since (f: T) in (f_i: T_i)*, then there exists some i’ for which
            f_i’ = f.
        + From canonical-rec lemma, we have:
            exists f_j* v_j*, e == {(f_j = v_j)*} /\
            forall i, exists v_i,
                (f_i = v_i) in (f_j = v_j)* /\
                [prog]; [] |- v_i : T_i
        + therefore, there exists v_i’ such that
                (f_i’ = v_i’) in (f_j = v_j)* /\
                [prog]; [] |- v_i’ : T_i’
        + thus, since (f_i’ = v_i’) in (f_j = v_j)*,
            and f_i’ = f,
            we show
                exists v_i’, [prog] |- {(f_j = v_j)*}.f ---> v_i’
                by R-RecProj.

        case 2.2: e is an instance of some named type
        + Since (f: T) in (f_i: T_i)*, then there exists some i' for which
            f_i' = f.
        + From canonical-rec lemma we have:
            exists f_j* v_j* a R,
            e == a.R({(f_j = v_j)*}) /\
            [prog]; [] |- a.R({(f_j = v_j)*}) : a.R /\
            [prog] |- a.R <: {(f_i: T_i)*}
        + Since [prog] |- a.R <: {(f_i: T_i)*}, and
            [prog]; [] |- a.R({(f_j = v_j)*}) : a.R
            then all fields f_i must have a corresponding value v_i
            in the instance definition a.R({(f_j = v_j)*}).
        + Thus, we know that exists some v_i', (f_i' = v_i') in (f_j = v_j)*
        + Thus, since (f_i' = v_i') in (f_j = v_j)*
            and f_i = f,
            we show that
            exists v_i', [prog] |- a.R({(f_j = v_j)*}).f ---> v_i'
                by R-InstProj.


CASE: T-Match

    []; [] |- p : T_p
    [prog]; [] |- match e with a.c {(f_arg = e_arg)*} : T'
    H1: [prog] |- a' ~> L
    H2: [prog]; [] |- e : a'.R
    H3: R = \overline{C_j {(f_i: T_i)*}} in L.ADTS
    H4: [prog]; [] |- a.c : {(f_arg: T_arg)*} -> {(C_j: {(f_i: T_i)*} -> T')*}
    H5: [prog]; [] |- {(f_arg = e_arg)*} : {(f_arg: T_arg)*}
    IND2: value(e) \/ exists e', [prog] |- e ---> e'
    IND5: value({(f_arg = e_arg)*}) \/ exists e'', [prog] |- {(f_arg = e_arg)*} ---> e''
    ----------------------------------------------

    case1: exists e', [prog] |- e ---> e'
        + thus, exists e',
            [prog] |- match e with a.c {(f_arg = e_arg)*} ---> match e' with a.c {(f_arg = e_arg)*}
            by R-MatchExp

    case2: value(e), exists e'', [prog] |- {(f_arg = e_arg)*} ---> e''
        + thus, exists e'',
            [prog] |- match v with a.c {(f_arg = e_arg)*} ---> match v with a.c e''
        by R-MatchCases

    case3: value(e), {(f_arg = v_arg)*}
        + We know that since value(e) and [prog]; [] |- e : a'.R,
            by Canonical-Fam
            e can either be an instance of a named record type,
            or an instance of an ADT.

        + Since and the program p is well-typed ([]; [] |- p : T_p),
            and that means all family definitions are well-formed,
            there can be no duplicate type names.

        + Since R has an ADT definition in L.ADTS,
            e must be an instance of an ADT and have shape a'.R(C {(f_k = v_k)*}).

        + thus, exists (a.c {(f_arg = v_arg)*}).C {(f_k = v_k)*},
            [prog] |- match a'.R(C {(f_k = v_k)*}) with a.c {(f_arg = v_arg)*} --->
            (a.c {(f_arg = v_arg)*}).C {(f_k = v_k)*}
            by R-MatchFinal.


CASE: T-If

    []; [] |- p : T_p
    [prog]; [] |- if e then g else g' : T'
    H1: [prog]; [] |- e : B
    H2: [prog]; [] |- g : T'
    H3: [prog]; [] |- g' : T'
    IND1: value(e) \/ exists e', [prog] |- e ---> e'
    ----------------------------------------------------

    case1: exists e', [prog] |- e ---> e'
        + thus,
            exists (if e' then g else g'),
            [prog] |- if e then g else g' ---> if e' then g else g'
            by R-IfGuard.

    case2: value(e)
        This means e is just a boolean expression b

        case 2.1: e = true
        + thus,
            exists g, [prog] |- if true then g else g' ---> g
            by R-IfTrue.

        case 2.1: e = false
        + thus,
            exists g', [prog] |- if false then g else g' ---> g'
            by R-IfFalse.
```

Preservation Nested

```
Statement:
    For any expression e in program p,
    []; [] |- p : T_p /\
    [prog]; [] |- e : T' /\
    (exists e', [prog] |- e ---> e')
    --> [prog]; [] |- e' : T'

Proof by induction on the typing derivation [prog]; [] |- e : T'.

The following are already included in the premise when necessary:
    - premises from induction on the typing derivation (named H1, H2, etc)
    - induction hypotheses (named IND1, IND2, etc)
        + for induction hypotheses, we write (IND: H' --> H'') as shorthand when
        the full hypothesis is (IND: (H1 /\ ... /\ Hn /\ H' --> H'')
        and H1 ... Hn are satisfied according to the context.

CASE: T-Num

    []; [] |- p : T_p
    [prog]; [] |- n: N
    (exists e', [prog] |- n ---> e')
    -------------
    Contradiction, n is a value and cannot reduce.


CASE: T-Bool

    []; [] |- p : T_p
    [prog]; [] |- b: B
    (exists e', [prog] |- b ---> e')
    -------------
    Contradiction, b is a value and cannot reduce.


CASE: T-Var

    []; [] |- p : T_p
    [prog]; [] |- x : T'
    (exists e', [prog] |- x ---> e')
    H1: x:T \in []
    -------------
    Contradiction in H1


CASE: T-Lam

    []; [] |- p : T_p
    [prog]; [] |- lam (x : T). e : T -> T'
    (exists e', [prog] |- lam (x : T). e ---> e')
    -----------------------------------
    Contradiction, lam is a value and cannot reduce.


CASE: T-App

    []; [] |- p : T_p
    [prog]; [] |- g e : T'
```

```
(exists e'', [prog] |- g e ---> e'')

H1: [prog]; [] |- e : T
H2: [prog]; [] |- g : T -> T'

IND1: (exists e', [prog] |- e ---> e') --> [prog]; [] |- e' : T
IND2: (exists g', [prog] |- g ---> g') --> [prog]; [] |- g' : T -> T'
------------------------

We know that the expression (g e) can reduce.
The following cases are possible:

case1: g can be reduced, R-App applies
    + Then we have
        [prog] |- g e ---> g' e
        and [prog] |- g ---> g' for some g'.
    + need to show that
        [prog]; [] |- g' e : T'
    + Since we have
        [prog]; [] |- g' : T -> T' by IND2, and
        [prog]; [] |- e : T by H1,
        we derive
        [prog]; [] |- g' e : T' by T-App.

case2: e can be reduced, R-LamArg applies
    + R-LamArg applies with g = v:
        [prog] |- v e ---> v e' and
        [prog] |- e ---> e' for some e'
    + need to show that
        [prog]; [] |- v e' : T'
    + Since we have
        [prog]; [] |- e' : T by IND1, and
        [prog]; [] |- g : T -> T' by H2,
        and we also know that g = v,
        we derive
        [prog]; [] |- v e' : T' by T-App.

case3: value(g), value(e)
    + by Canonical-Fun, we have
        exists x S e''', [prog] |- T <: S /\ g == (lam (x: S). e''')
        since g is a value with an arrow type.
    + R-LamApply applies with g = lam (x: S). e''', and e = v:
        [prog] |- (lam (x: S). e''') v  -->  [x:=v] e'''
    + need to show that
        [prog]; [] |- [x:=v] e''' : T'
    + Since we know that
        [prog]; [] |- (lam (x: S). e''') : T -> T' by H2 and Canonical-Fun,
        we also know
        [prog] |- WF(T) and
        [prog]; (x:S, []) |- e''' : T'
        by inversion (T-Lam).
    + we also know that
        [prog]; [] |- v : T by H1,
        and since [prog] |- T <: S we derive
        [prog]; [] |- v : S by T-Subs.
    + Since we have [prog]; (x:S, []) |- e''' : T' and
        [prog]; [] |- v : S,
        we derive
        [prog]; [] |- [x:=v] e''' : T'
        by our lemma that substitution is type-preserving.
```

```
CASE: T-Rec

    []; [] |- p : T_p
    [prog]; [] |- {(f_i = e_i)*} : {(f_i: T_i)*}
    (exists e', [prog] |- {(f_i = e_i)*} ---> e')

    forall i, Hi:
        [prog]; [] |- e_i : T_i

    forall i, INDi:
        (exists e_i', [prog] |- e_i ---> e_i') --> [prog]; [] |- e_i' : T_i
    -------------------------------------

    Need to show [prog]; [] |- e' : {(f_i: T_i)*}.
    By premise, we know that the record expression can reduce
        (the case where all e_i's are values is a contradiction).
    Consider e_i in order from left to right,
    there exists some i such that e_i is the first that can reduce.
    + R-Rec applies:
        [prog] |- {f_0 = v_0, ..., f_i-1 = v_i-1, f_i = e_i, ...} --->
        {f_0 = v_0, ..., f_i-1 = v_i-1, f_i = e_i', ...}
    + Now need to show that
        [prog]; [] |- {f_0 = v_0, ..., f_i-1 = v_i-1, f_i = e_i', ...} : {(f_i: T_i)*}
    + Since we know that [prog]; [] |- e_i' : T_i by INDi,
        and for all other i we have Hi: [prog]; [] |- e_i : T_i by premise, we derive that
        [prog]; [] |- {f_0 = v_0, ..., f_i-1 = v_i-1, f_i = e_i', ...} : {(f_i: T_i)*}
        (only one field has reduced, but the new reduced expression e_i'
        still has the same type T_i).



CASE: T-Proj

    []; [] |- p : T_p
    [prog]; [] |- e.f : T'
    (exists e', [prog] |- e.f ---> e')

    H1: [prog]; [] |- e : {(f_i: T_i)*}
    H2: (f: T') in (f_i: T_i)*

    IND1: (exists e'', [prog] |- e ---> e'') --> [prog]; [] |- e'' : {(f_i: T_i)*}
    -------------------------------------

    By premise, the expression e.f can reduce.
    The following cases are possible:

    case1: e can reduce
        + then R-Proj applies
        and we know there exists some e'' such that
            [prog] |- e.f ---> e''.f and
            [prog] |- e ---> e'' by R-Proj.
        + Need to show that [prog]; [] |- e''.f : T'
        + Since we know by IND1 that [prog]; [] |- e'' : {(f_i: T_i)*},
            and by H2 we have (f: T') in (f_i: T_i)*,
            we derive [prog]; [] |- e''.f : T' by T-Proj.

    case2: e is a value v with a record type, by Canonical-Rec there are 2 cases:
        case 2.1: e is a record expression
        + by Canonical-Rec:
            exists f_j* v_j*, v == {(f_j = v_j)*} /\
            forall i, (exists v_i, (f_i = v_i) in (f_j = v_j)* /\ [prog]; [] |- v_i : T_i)
        + Then the reduction is an instance of R-RecProj
        and there exists some v' such that:
            [prog] |- {(f_j = v_j)*}.f ---> v' and
            (f = v') in (f_j = v_j)* by R-RecProj.
        + Need to show that [prog]; [] |- v' : T'.
        + Since we have
            (f = v') in (f_j = v_j)* and
            (f: T') in (f_i: T_i)*,
            f refers to the same field in both,
            there are no duplicate fields by convention,
            and each field f_i appears in
            (f_j = v_j)* with some value that has the corresponding type T_i,
            we know that [prog]; [] |- v' : T'.

        case 2.2: e is an instance expression
        + exists f_j* v_j* a R,
            v == a.R({(f_j = v_j)*}) /\
            [prog]; [] |- a.R({(f_j = v_j)*}) : a.R /\
            [prog] |- a.R <: {(f_i: T_i)*})
        + Then the reduction is an instance of R-InstProj and
            there exists some v' such that:
            [prog] |- a.R({(f_j = v_j)*}).f ---> v' and
            (f = v') in (f_j = v_j)* by R-InstProj.
        + Need to show that [prog]; [] |- v' : T'
        + Since we have
            (f = v') in (f_j = v_j)* and
            (f: T') in (f_i: T_i)*,
            f refers to the same field in both,
            there are no duplicate fields by convention,
            and each field f_i appears in
            (f_j = v_j)* with some value that has the corresponding type T_i,
            we know that [prog]; [] |- v' : T'.


CASE: T-FamFun

    []; [] |- p : T_p
    [prog]; [] |- a.m : T -> T'
    (exists e', [prog] |- a.m ---> e')
    ---------------------------------------------

    + the reduction is an instance of R-FamFun so there exists some
        lam (x: T).e such that
        [prog] |- a.m ---> lam (x: T).e
        [prog] |- a ~> L and
        m: (T -> T') = lam (x: T). e in L.FUNS
        by R-FamFun.
    + need to show that [prog]; [] |- lam (x: T).e : T -> T'
    + Since the program is well-typed ([]; [] |- p : T_p),
        all nested family definitions are well-formed,
        and well-formedness of definitions is preserved by concatenation,
        the definition of m in the linkage must be well-typed.


CASE: T-Cases
```

```
    []; [] |- p : T_p
    [prog]; [] |- a.c : {(f_i:T_i)*} -> {(C_j:T_j->T_j')*}
    (exists e', [prog] |- a.c ---> e')
    ---------------------------------------------

    + the reduction is an instance of R-Cases so we have:
        [prog] |- a.c ---> lam (x: {(f_i:T_i)*}). {(C_j = lam (y_j: T_j). e_j)*}
        [prog] |- a ~> L
        c <a'.R> : {(f_i:T_i)*} -> {(C_j:T_j->T_j')*} =
            lam (x: {(f_i:T_i)*}). {(C_j = lam (y_j: T_j). e_j)*} in L.CASES
        by R-Cases.
    + need to show that
        [prog]; [] |- lam (x: {(f_i:T_i)*}). {(C_j = lam (y_j: T_j). e_j)*} :
        {(f_i:T_i)*} -> {(C_j:T_j->T_j')*}
    + Since the program is well-typed ([]; [] |- p : T_p),
        all nested family definitions are well-formed,
        and well-formedness of definitions is preserved by concatenation,
        the definition of c in the linkage must be well-typed.


CASE: T-Constr

    []; [] |- p : T_p
    [prog]; [] |- a.R({(f_i = e_i)*}) : a.R
    (exists e', [prog] |- a.R({(f_i = e_i)*}) ---> e')

    H1: [prog] |- a ~> L
    H2: R = {(f_i: T_i)*} in L.TYPES
    forall i, Hi:
        [prog]; [] |- e_i : T_i

    forall i, INDi:
        (exists e_i', [prog] |- e_i ---> e_i') --> [prog]; [] |- e_i' : T_i
    -----------------------------------

    + By premise, we know that a.R({(f_i = e_i)*}) can reduce.
        The only rule that can apply is R_Instance.
    + Thus, by R_Instance, we have:
        [prog] |- a.R({(f_i = e_i)*}) ---> a.R(e'') for some e'',
        and also that
        [prog] |- {(f_i = e_i)*} ---> e''.
    + Since we only have one reduction rule for records, R-Rec,
        it must be true that e'' is also some record, {(f_i = e_i')*}
        with the same fields,
        and that there exists some index j such that e_j is the first
        expression in the record {(f_i = e_i)*} that can reduce,
        and [prog] |- e_j ---> e_j'.
    + Show that [prog]; [] |- a.R({(f_i = e_i')*}) : a.R
    + Since we know by R-Rec that
        [prog] |- {(f_i = e_i)*} ---> {(f_i = e_i')*}, and
        that e_j is the first field that reduces ([prog] |- e_j ---> e_j'), we know that
        the two records are identical except for the single field that reduces, e_j.
    + By INDi, e_j' has the same type as e_j in the original record.
    + Thus, we know that forall i,
        [prog]; [] |- e_i' : T_i
        (because in the reduced record {(f_i = e_i')*} all expressions e_i' except
         e_j stay unchanged, and e_j reduces to e_j' with the same type as e_j).
    + From premises
        [prog] |- a ~> L and
        R = {(f_i: T_i)*} in L.TYPES
        and because we showed
        forall i, [prog]; [] |- e_i' : T_i
        we derive
        [prog]; [] |- a.R({(f_i = e_i')*}) : a.R by rule T-Constr.



CASE: T-ADT

    []; [] |- p : T_p
    [prog]; [] |- a.R(C {(f_k = e_k)*}) : a.R
    (exists e', [prog] |- a.R(C {(f_k = e_k)*}) ---> e')

    H1: [prog] |- a ~> L
    H2: R = \overline{C_j {(f_i: T_i)*}} in L.ADTS
    H3: C {(f_k: T_k)*} in \overline{C_j {(f_i: T_i)*}}
    forall k, Hk:
        [prog]; [] |- e_k : T_k

    forall k, INDk:
        (exists e_k', [prog] |- e_k ---> e_k') --> [prog]; [] |- e_k' : T_k
    --------------------------------------------

    + By premise, we know that a.R(C {(f_k = e_k)*}) can reduce.
        The only rule that can apply is R_ADT.
    + Thus, by R_ADT, we have:
        [prog] |- a.R(C {(f_k = e_k)*}) ---> a.R(C e'') for some e'',
        and also that
        [prog] |- {(f_k = e_k)*} ---> e''.
    + Since we only have one reduction rule for records, R-Rec,
        it must be true that e'' is also some record, {(f_k = e_k')*}
        with the same fields,
        and that there exists some index j such that e_j is the first
        expression in the record {(f_k = e_k)*} that can reduce,
        and [prog] |- e_j ---> e_j'.
    + Show that [prog]; [] |- a.R(C {(f_k = e_k')*}) : a.R
    + Since we know by R-Rec that
        [prog] |- {(f_k = e_k)*} ---> {(f_k = e_k')*}, and
        that e_j is the first field that reduces ([prog] |- e_j ---> e_j'), we know that
        the two records are identical except for the single field that reduces, e_j.
    + By INDk, e_j' has the same type as e_j in the original record.
    + Thus, we know that forall k,
        [prog]; [] |- e_k' : T_k
        (because in the reduced record {(f_k = e_k')*} all expressions e_k' except
         e_j stay unchanged, and e_j reduces to e_j' with the same type as e_j).
    + From premises
        [prog] |- a ~> L
        R = \overline{C_j {(f_i: T_i)*}} in L.ADTS
        C {(f_k: T_k)*} in \overline{C_j {(f_i: T_i)*}}
        and because we showed forall k,
        [prog]; [] |- e_k' : T_k,
        we can derive
        [prog]; [] |- a.R({(f_k = e_k')*}) : a.R by rule T-ADT.



CASE: T-Match

    []; [] |- p : T_p
    [prog]; [] |- match e with a.c {(f_arg = e_arg)*} : T'
```

```
(exists e'', [prog] |- match e with a.c {(f_arg = e_arg)*} ---> e'')

H1: [prog] |- a' ~> L
H2: [prog]; [] |- e : a'.R
H3: R = \overline{C_j {(f_i: T_i)*}} in L.ADTS
H4: [prog]; [] |- a.c : {(f_arg: T_arg)*} -> {(C_j: {(f_i: T_i)*} -> T')*}
H5: [prog]; [] |- {(f_arg = e_arg)*} : {(f_arg: T_arg)*}

INDe: (exists e', [prog] |- e ---> e') --> [prog]; [] |- e' : a'.R
INDarg: (exists e'_arg*, [prog] |- {(f_arg = e_arg)*} ---> {(f_arg = e'_arg)*}) --->
        [prog]; [] |- {(f_arg = e'_arg)*} : {(f_arg: T_arg)*}
----------------------------------------------

By premise, we know that the match expression can reduce.
The following cases are possible:

case1: R-MatchExp applies:
    + then, we have
        [prog] |- match e with a.c {(f_arg = e_arg)*} ---> match e' with a.c {(f_arg = e_arg)*}
        and [prog] |- e ---> e' for some e'
    + show that [prog]; [] |- match e' with a.c {(f_arg = e_arg)*} : T'
    + Since we know that
        [prog]; [] |- e' : a'.R by INDe,
        and also H1, H3, H4, H5
        we derive
        [prog]; [] |- match e' with a.c {(f_arg = e_arg)*} : T' by T-Match.

case2: R-MatchCases applies
    + then, we have e = v and some e'_arg* such that:
        [prog] |- match v with a.c {(f_arg = e_arg)*} ---> match v with a.c {(f_arg = e'_arg)*}
        and
        [prog] |- {(f_arg = e_arg)*} ---> {(f_arg = e'_arg)*}
    + Show that [prog]; [] |- match v with a.c {(f_arg = e'_arg)*} : T'
    + Since we know that
        [prog]; [] |- v : a'.R by H2 because e = v
        and
        [prog]; [] |- {(f_arg = e'_arg)*} : {(f_arg: T_arg)*} by INDarg,
        and also H1, H3, H4 we derive
        [prog]; [] |- match v with a.c {(f_arg = e'_arg)*} : T' by T-Match.


case3: R-MatchFinal applies
    + then, we have
        e = a'.R(C {(f_k = v_k)*}) for some C, f_k*, v_k*
        and {(f_arg = e_arg)*} = {(f_arg = v_arg)*} (all values)
        and
        [prog] |- match a'.R(C {(f_k = v_k)*}) with a.c {(f_arg = v_arg)*}  --->
        (a.c {(f_arg = v_arg)*}).C {(f_k = v_k)*}
    + We must now show that
        [prog]; [] |- (a.c {(f_arg = v_arg)*}).C {(f_k = v_k)*} : T'
    + Since we have
        H4: [prog]; [] |- a.c : {(f_arg: T_arg)*} -> {(C_j: {(f_i: T_i)*} -> T')*}
        H5: [prog]; [] |- {(f_arg = e_arg)*} : {(f_arg: T_arg)*}
        we derive
        [prog]; [] |- a.c {(f_arg = e_arg)*}: {(C_j: {(f_i: T_i)*} -> T')*} by T-App.
    + We also know that since
        [prog]; [] |- a'.R(C {(f_k = v_k)*}) : a'.R
        by H2 (with e = a'.R(C {(f_k = v_k)*}))
        by rule T-ADT we also have the following:
        C {(f_k : T_k)*} in \overline{C_j {(f_i: T_i)*}} and
```

```
          forall k,
              [prog]; [] |- v_k : T_k
          (the latter is also equivalent to
              [prog]; [] |- {(f_k = v_k)*} : {(f_k : T_k)*})
      + Since we have
          C {(f_k : T_k)*} in \overline{C_j {(f_i: T_i)*}}
          and since by convention the j and i indexes in the definition of R
          and the type of a.c range over the same variables,
          we have
          (C : {(f_k : T_k)*} -> T') in {(C_j: {(f_i: T_i)*} -> T')*}
      + Since we have
          [prog]; [] |- a.c {(f_arg = e_arg)*}: {(C_j: {(f_i: T_i)*} -> T')*}
          and (C : {(f_k : T_k)*} -> T') in {(C_j: {(f_i: T_i)*} -> T')*},
          we derive
          [prog]; [] |- (a.c {(f_arg = e_arg)*}).C : {(f_k: T_k)*} -> T' by rule T-Proj.
      + Since
          [prog]; [] |- (a.c {(f_arg = e_arg)*}).C : {(f_k: T_k)*} -> T' and
          [prog]; [] |- {(f_k = v_k)*} : {(f_k : T_k)*}
          we derive
          [prog]; [] |- (a.c {(f_arg = e_arg)*}).C {(f_k = v_k)*} : T' by rule T-App.


CASE: T-Subs

    []; [] |- p : T_p
    [prog]; [] |- e : T'
    (exists e', [prog] |- e ---> e')

    H1: [prog]; [] |- e : T
    H2: [prog] |- T <: T'

    IND1: (exists e', [prog] |- e ---> e') --> [prog]; [] |- e' : T
    --------------------

    need to show [prog]; [] |- e' : T'.
    Since we have [prog]; [] |- e' : T by IND, and [prog] |- T <: T',
        we derive [prog]; [] |- e' : T' by T-Subs.


CASE: T-If

    []; [] |- p : T_p
    [prog]; [] |- if e then g else g' : T'
    (exists e'', [prog] |- if e then g else g' ---> e'')

    H1: [prog]; [] |- e : B
    H2: [prog]; [] |- g : T'
    H3: [prog]; [] |- g' : T'

    IND1: (exists e', [prog] |- e ---> e') --> [prog]; [] |- e : B
    --------------------------------------------------------------

    By premise, the if expression can be reduced. there are 3 cases:

    case1: R-IfGuard applies
    + Then, [prog] |- if e then g else g' ---> if e' then g else g'
        and [prog] |- e ---> e' for some e'.
    + by IND1, we have [prog]; [] |- e' : B
    + From [prog]; [] |- e' : B and premises H2 and H3, we derive
        [prog]; [] |- if e' then g else g' by T-If.
```

```
case2: R-IfTrue applies
+ Then, [prog] |- if true then g else g' ---> g
+ [prog]; [] |- g : T' by H2

case 3: R-IfFalse applies
+ Then, [prog] |- if false then g else g' ---> g'
+ [prog]; [] |- g' : T' by H3
```

Linkage Well-Formedness after Parsing

```
Lemma: Parse-Prog-WF-Linkage:

    []; [] |- p: T /\
    parse(p) = L -->
    [] |- WF(L)

Proof:
    + from rule PARSE-PROG, L.self = prog, and prog has no ancestors.
    + Since we have a well-typed program, we know that each nested family
        definition in p is well formed by rule T-Prog:
            forall i, [prog] |- WF(famdef_i)

        For each famdef_i, we know by Parse-Famdef-WF-Linkage that
        since each famdef_i is well-formed, the corresponding parsed linkage
        is also well-formed:

        forall i,
            []; [] |- p: T /\
            [prog] |- WF(famdef_i) /\
            parse(prog, famdef_i) = L_i -->
            [prog] |- WF(L_i)

        Thus, for all mappings A_i |-> L_i in L.NEST, L_i is well-formed.
    + Linkage L does not have any other stored elements by rule parse-prog,
      thus we've shown [] |- WF(L).


Lemma: Parse-Famdef-WF-Linkage:

    []; [] |- p: T /\
    sp :: K |- WF(famdef) /\
    parse(sp, famdef) = L -->
    sp :: K |- WF(L)

Proof:
    + Since we know that
        sp :: K |- WF(famdef),
        by rule WF-FamDef we know that the path to this family A,
        self(sp.A), is not an ancestor of itself.
    + Each nested linkage is well-formed by induction on
        sp :: K |- WF(famdef).
    + For each type definition
        type R (+)?= {(f_i: T_i (= v_i)?)*} in famdef,
        by rule PARSE-FAMDEF we know that the linkage L
        has two corresponding entries:
            R_((+)?=) |-> {(f_i: T_i)*} in L.TYPES
            - this mapping is WF because famdef is WF, by first premise of rule WF-TypDef
                (the record type is WF)
        and R_((+)?=) |-> {(f_i: v_i)*} in L.DEFAULTS
            - this mapping is WF by the second premise of rule WF-TypDef
            (all v_i's typecheck) and we know that the fields that have defaults are a subset of
            all fields of R, because they are being pulled from the same type definition
            in famdef (and some fields may omit defaults).
        thus, all type mappings and default mappings in L are well-formed.
    + For each ADT definition
        R_((+)?=) |-> \overline{C_j {(f_i: T_i)*}} in L.ADTS
        - this mapping is WF because famdef is WF, and by the premise of rule
        WF-AdtDef forall constructors C_j the corresponding inputs record type
        is well-formed.
    + For each function definition
        m |-> (T->T', lam(x: T).e) in L.FUNS
        - this mapping is WF because famdef is WF and by the premise of rule
        WF-FunDef we have that the arrow type is WF and the body is well-typed.
    + For each cases definition
        c_((+)?=) |-> (<a.R>, T->T', lam(x: T).e) in L.CASES
        - this mapping is WF because famdef is WF and by the premise of rule
        WF-Cases we have that the arrow type is WF and the body is well-typed.
    + Thus, we have that L is WF.
```

Concatenation of well-formed linkages

```
Lemma: concat-WF-linkage

    K |- WF(L1) /\ K |- WF(L2) /\
    L1 + L2 = L3 -->
    K |- WF(L3)

Proof.
    + Since L3.self and L3.super come directly from L2 which is WF,
        we know there's no circular inheritance.
    + show WF(NEST1) + WF(NEST2) = WF(NEST3)
        two cases in rule CAT-NEST:
        ++ all linkages in the disjoint union are WF because
        they're coming from WF(NEST1) and WF(NEST2).
        ++ every pair of linkages L and L' from NEST1 and
        NEST2 are well-formed, and their concatenation results in
        a well-formed linkage by induction.
    + show WF(TYPES1) + WF(TYPES2) = WF(TYPES3)
        two cases in rule CAT-TYPES:
        ++ all types in the disjoint union are WF because
        they're coming from WF(TYPES1) and WF(TYPES2).
        ++ all extended record types are WF because the two record
        types being concatenated are coming from WF(TYPES1) and WF(TYPES2),
        concatenation of record types does not allow duplicate fields, and
        no duplicate type name is introduced since the type name R
        is kept.
    + similar reasoning for WF(DEFS1) + WF(DEFS2) = WF(DEFS3)
        using rule CAT-DEFS
    + similar reasoning for WF(ADTS1) + WF(ADTS2) = WF(ADTS3)
        using rule CAT-ADTS, with the added assurance that
        concatenation of ADT definitions does not result in duplicate
        constructors or duplicate record fields.
    + show WF(FUNS1) + WF(FUNS2) = WF(FUNS3)
        two cases in rule CAT-FUNS:
        + all functions in the disjoint union are WF because they're coming
        from WF(FUNS1) and WF(FUNS2)
        + overwriting case: the function name, type and body are
        coming from FUNS2 which is well-formed, so the resulting definition
        is well formed.
    + show WF(CASES1) + WF(CASES2) = WF(CASES3)
        three cases in rule CAT-CASES:
        + all cases in the disjoint union are WF because they're coming
        from WF(CASES1) and WF(CASES2)
        + overwriting case: the case match type, type, and body are coming
```

from CASES2 which is well-formed, so the resulting definition is
well-formed.
+ extension case:
all extended cases definitions are WF because:

- the inherited part is coming from CASES1 which is WF,
  which means type T'_1 names all inherited constructors and provides
  handlers for those constructors that typecheck.
- Since CASES2 is WF, Type T'_2 names all new constructors and
  provides handlers for those constructors that typecheck.
- type T' (the output type of the combined definition) is just a
  record concatenation, meaning it covers all constructors inherited
  and new. The rule implicitly checks that no duplicates result in this
  record concatenation. Thus, the combined type T' is well-formed.
- the resulting body of the cases is just a record concatenation of the
  inherited and extended bodies (with the potentially conflicting lambda
  variable for match context replaced with a fresh one in both bodies).
  The rule implicitly checks that no duplicates result in this record concatenation.
- thus, the resulting cases body typechecks according to the resulting
  cases type by rule T-Lam. The combined body e has the combined record type T'.
  thus, each combined cases construct is well-formed.