A Solution to the PoplMark Challenge in Isabelle/HOL

Stefan Berghofer

March 17, 2025

Abstract

We present a solution to the PoplMark challenge designed by Aydemir et al., which has as a goal the formalization of the meta-theory of System $F_{<:}$. The formalization is carried out in the theorem prover Isabelle/HOL using an encoding based on de Bruijn indices. We start with a relatively simple formalization covering only the basic features of System $F_{<:}$, and explain how it can be extended to also cover records and more advanced binding constructs.

Contents

1 General Utilities

This section introduces some general utilities that will be useful later on in the formalization of System $F_{<:}$.

The following rewrite rules are useful for simplifying mutual induction rules.

lemma *True-simps*:
  $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
  $(PROP\ P \Longrightarrow True) \equiv PROP\ Trueprop\ True$
  $(\bigwedge x.\ True) \equiv PROP\ Trueprop\ True$
  by *auto*

Unfortunately, the standard introduction and elimination rules for bounded universal and existential quantifier do not work properly for sets of pairs.

lemma *ballpI*: $(\bigwedge x\ y.\ (x,\ y) \in A \Longrightarrow P\ x\ y) \Longrightarrow \forall (x,\ y) \in A.\ P\ x\ y$
  by *blast*

lemma *bpspec*: $\forall (x,\ y) \in A.\ P\ x\ y \Longrightarrow (x,\ y) \in A \Longrightarrow P\ x\ y$
  by *blast*

lemma *ballpE*: $\forall (x,\ y) \in A.\ P\ x\ y \Longrightarrow (P\ x\ y \Longrightarrow Q) \Longrightarrow$
  $((x,\ y) \notin A \Longrightarrow Q) \Longrightarrow Q$
  by *blast*

lemma *bexpI*: $P\ x\ y \Longrightarrow (x,\ y) \in A \Longrightarrow \exists (x,\ y) \in A.\ P\ x\ y$
  by *blast*

lemma *bexpE*: $\exists (x,\ y) \in A.\ P\ x\ y \Longrightarrow$
  $(\bigwedge x\ y.\ (x,\ y) \in A \Longrightarrow P\ x\ y \Longrightarrow Q) \Longrightarrow Q$
  by *blast*

lemma *ball-eq-sym*: $\forall (x,\ y) \in S.\ f\ x\ y = g\ x\ y \Longrightarrow \forall (x,\ y) \in S.\ g\ x\ y = f\ x\ y$
  by *auto*

lemma *wf-measure-size*: *wf* (*measure size*) **by** *simp*

notation
  Some (‹⌊-⌋›)

notation
  None (‹⊥›)

notation
  length (‹‖-‖›)

notation
  Cons (‹- ::/ -› [*66*, *65*] *65*)

The following variant of the standard *nth* function returns ⊥ if the index is out of range.

primrec
  nth-el :: *'a list* ⇒ *nat* ⇒ *'a option* (‹-⟨-⟩› [*90*, *0*] *91*)
where
  []⟨*i*⟩ = ⊥
| (*x* # *xs*)⟨*i*⟩ = (*case i of 0* ⇒ ⌊*x*⌋ | *Suc j* ⇒ *xs* ⟨*j*⟩)

lemma *nth-el-append1* [*simp*]: *i* < ‖*xs*‖ ⟹ (*xs* @ *ys*)⟨*i*⟩ = *xs*⟨*i*⟩
proof (*induct xs arbitrary*: *i*)
  case *Nil*
  then show *?case*
    by *simp*
next
  case (*Cons a xs i*)
  then show *?case* **by** (*cases i*) *auto*
qed

lemma *nth-el-append2* [*simp*]: ‖*xs*‖ ≤ *i* ⟹ (*xs* @ *ys*)⟨*i*⟩ = *ys*⟨*i* − ‖*xs*‖⟩
proof (*induct xs arbitrary*: *i*)
  case *Nil*
  then show *?case*
    by *simp*
next
  case (*Cons a xs i*)
  then show *?case* **by** (*cases i*) *auto*
qed

Association lists

primrec *assoc* :: (*'a* × *'b*) *list* ⇒ *'a* ⇒ *'b option* (‹-⟨-⟩?› [*90*, *0*] *91*)
where
  []⟨*a*⟩? = ⊥
| (*x* # *xs*)⟨*a*⟩? = (*if fst x* = *a then* ⌊*snd x*⌋ *else xs*⟨*a*⟩?)

primrec *unique* :: (*'a* × *'b*) *list* ⇒ *bool*
where
  unique [] = *True*
| *unique* (*x* # *xs*) = (*xs*⟨*fst x*⟩? = ⊥ ∧ *unique xs*)

lemma *assoc-set*: *ps*⟨*x*⟩? = ⌊*y*⌋ ⟹ (*x*, *y*) ∈ *set ps*
  by (*induct ps*) (*auto split*: *if-split-asm*)

lemma *map-assoc-None* [*simp*]:
  ps⟨*x*⟩? = ⊥ ⟹ *map* (λ(*x*, *y*). (*x*, *f x y*)) *ps*⟨*x*⟩? = ⊥
  by (*induct ps*) *auto*

no-syntax
  -Map :: *maplets* => *'a* ⇀ *'b* (‹(‹*indent*=*1 notation*=‹*mixfix map*››[-])›)

2 Formalization of the basic calculus

In this section, we describe the formalization of the basic calculus without records. As a main result, we prove *type safety*, presented as two separate theorems, namely *preservation* and *progress*.

2.1 Types and Terms

The types of System $F_{<:}$ are represented by the following datatype:

datatype *type* =
  TVar nat
| *Top*
| *Fun type type* (**infixr** ‹→› *200*)
| *TyAll type type* (‹(3∀<:-./ -)› [*0*, *10*] *10*)

The subtyping and typing judgements depend on a *context* (or environment) Γ containing bindings for term and type variables. A context is a list of bindings, where the ith element $\Gamma\langle i\rangle$ corresponds to the variable with index i.

datatype *binding* = *VarB type* | *TVarB type*
type-synonym *env* = *binding list*

In contrast to the usual presentation of type systems often found in textbooks, new elements are added to the left of a context using the *Cons* operator :: for lists. We write *is-TVarB* for the predicate that returns *True* when applied to a type variable binding, function *type-ofB* extracts the type contained in a binding, and *mapB f* applies *f* to the type contained in a binding.

primrec *is-TVarB* :: *binding* ⇒ *bool*
where
  is-TVarB (*VarB T*) = *False*
| *is-TVarB* (*TVarB T*) = *True*

primrec *type-ofB* :: *binding* ⇒ *type*
where
  type-ofB (*VarB T*) = *T*
| *type-ofB* (*TVarB T*) = *T*

primrec *mapB* :: (*type* ⇒ *type*) ⇒ *binding* ⇒ *binding*
where
  mapB f (*VarB T*) = *VarB* (*f T*)
| *mapB f* (*TVarB T*) = *TVarB* (*f T*)

The following datatype represents the terms of System $F_{<:}$:

datatype *trm* =
  Var nat

| *Abs type trm* (‹(3λ:-./ -)› [0, 10] 10)
| *TAbs type trm* (‹(3λ<:-./ -)› [0, 10] 10)
| *App trm trm* (**infixl** ‹·› 200)
| *TApp trm type* (**infixl** ‹$\cdot_\tau$› 200)

2.2 Lifting and Substitution

One of the central operations of λ-calculus is *substitution*. In order to avoid that free variables in a term or type get "captured" when substituting it for a variable occurring in the scope of a binder, we have to increment the indices of its free variables during substitution. This is done by the lifting functions $\uparrow_\tau\ n\ k$ and $\uparrow\ n\ k$ for types and terms, respectively, which increment the indices of all free variables with indices $\geq k$ by n. The lifting functions on types and terms are defined by

primrec *liftT* :: *nat* ⇒ *nat* ⇒ *type* ⇒ *type* (‹$\uparrow_\tau$›)
where
$\uparrow_\tau\ n\ k\ (TVar\ i) = (if\ i < k\ then\ TVar\ i\ else\ TVar\ (i + n))$
$|\ \uparrow_\tau\ n\ k\ Top = Top$
$|\ \uparrow_\tau\ n\ k\ (T \rightarrow U) = \uparrow_\tau\ n\ k\ T \rightarrow \uparrow_\tau\ n\ k\ U$
$|\ \uparrow_\tau\ n\ k\ (\forall <:T.\ U) = (\forall <:\uparrow_\tau\ n\ k\ T.\ \uparrow_\tau\ n\ (k + 1)\ U)$

primrec *lift* :: *nat* ⇒ *nat* ⇒ *trm* ⇒ *trm* (‹↑›)
where
$\uparrow n\ k\ (Var\ i) = (if\ i < k\ then\ Var\ i\ else\ Var\ (i + n))$
$|\ \uparrow n\ k\ (\lambda:T.\ t) = (\lambda:\uparrow_\tau\ n\ k\ T.\ \uparrow n\ (k + 1)\ t)$
$|\ \uparrow n\ k\ (\lambda<:T.\ t) = (\lambda<:\uparrow_\tau\ n\ k\ T.\ \uparrow n\ (k + 1)\ t)$
$|\ \uparrow n\ k\ (s \cdot t) = \uparrow n\ k\ s \cdot \uparrow n\ k\ t$
$|\ \uparrow n\ k\ (t \cdot_\tau T) = \uparrow n\ k\ t \cdot_\tau \uparrow_\tau\ n\ k\ T$

It is useful to also define an "unlifting" function $\downarrow_\tau\ n\ k$ for decrementing all free variables with indices $\geq k$ by n. Moreover, we need several substitution functions, denoted by $T[k \mapsto_\tau S]_\tau$, $t[k \mapsto_\tau S]$, and $t[k \mapsto s]$, which substitute type variables in types, type variables in terms, and term variables in terms, respectively. They are defined as follows:

primrec *substTT* :: *type* ⇒ *nat* ⇒ *type* ⇒ *type* (‹$\_[\_ \mapsto_\tau \_]_\tau$› [300, 0, 0] 300)
where
$(TVar\ i)[k \mapsto_\tau S]_\tau =$
$\quad (if\ k < i\ then\ TVar\ (i - 1)\ else\ if\ i = k\ then\ \uparrow_\tau\ k\ 0\ S\ else\ TVar\ i)$
$|\ Top[k \mapsto_\tau S]_\tau = Top$
$|\ (T \rightarrow U)[k \mapsto_\tau S]_\tau = T[k \mapsto_\tau S]_\tau \rightarrow U[k \mapsto_\tau S]_\tau$
$|\ (\forall <:T.\ U)[k \mapsto_\tau S]_\tau = (\forall <:T[k \mapsto_\tau S]_\tau.\ U[k+1 \mapsto_\tau S]_\tau)$

primrec *decT* :: *nat* ⇒ *nat* ⇒ *type* ⇒ *type* (‹$\downarrow_\tau$›)
where
$\downarrow_\tau\ 0\ k\ T = T$
$|\ \downarrow_\tau\ (Suc\ n)\ k\ T = \downarrow_\tau\ n\ k\ (T[k \mapsto_\tau Top]_\tau)$

primrec *subst* :: *trm* ⇒ *nat* ⇒ *trm* ⇒ *trm* (‹-[- ↦ -]› [*300*, *0*, *0*] *300*)
where
$(Var\ i)[k \mapsto s] = (if\ k < i\ then\ Var\ (i - 1)\ else\ if\ i = k\ then\ \uparrow k\ 0\ s\ else\ Var\ i)$
$|\ (t \cdot u)[k \mapsto s] = t[k \mapsto s] \cdot u[k \mapsto s]$
$|\ (t \cdot_\tau T)[k \mapsto s] = t[k \mapsto s] \cdot_\tau \downarrow_\tau 1\ k\ T$
$|\ (\lambda{:}T.\ t)[k \mapsto s] = (\lambda{:}\downarrow_\tau 1\ k\ T.\ t[k{+}1 \mapsto s])$
$|\ (\lambda{<:}T.\ t)[k \mapsto s] = (\lambda{<:}\downarrow_\tau 1\ k\ T.\ t[k{+}1 \mapsto s])$

primrec *substT* :: *trm* ⇒ *nat* ⇒ *type* ⇒ *trm* (‹-[- $\mapsto_\tau$ -]› [*300*, *0*, *0*] *300*)
where
$(Var\ i)[k \mapsto_\tau S] = (if\ k < i\ then\ Var\ (i - 1)\ else\ Var\ i)$
$|\ (t \cdot u)[k \mapsto_\tau S] = t[k \mapsto_\tau S] \cdot u[k \mapsto_\tau S]$
$|\ (t \cdot_\tau T)[k \mapsto_\tau S] = t[k \mapsto_\tau S] \cdot_\tau T[k \mapsto_\tau S]_\tau$
$|\ (\lambda{:}T.\ t)[k \mapsto_\tau S] = (\lambda{:}T[k \mapsto_\tau S]_\tau.\ t[k{+}1 \mapsto_\tau S])$
$|\ (\lambda{<:}T.\ t)[k \mapsto_\tau S] = (\lambda{<:}T[k \mapsto_\tau S]_\tau.\ t[k{+}1 \mapsto_\tau S])$

Lifting and substitution extends to typing contexts as follows:

primrec *liftE* :: *nat* ⇒ *nat* ⇒ *env* ⇒ *env* (‹$\uparrow_e$›)
where
$\uparrow_e n\ k\ [] = []$
$|\ \uparrow_e n\ k\ (B :: \Gamma) = mapB\ (\uparrow_\tau n\ (k + \|\Gamma\|))\ B :: \uparrow_e n\ k\ \Gamma$

primrec *substE* :: *env* ⇒ *nat* ⇒ *type* ⇒ *env* (‹-[- $\mapsto_\tau$ -]$_e$› [*300*, *0*, *0*] *300*)
where
$[][k \mapsto_\tau T]_e = []$
$|\ (B :: \Gamma)[k \mapsto_\tau T]_e = mapB\ (\lambda U.\ U[k + \|\Gamma\| \mapsto_\tau T]_\tau)\ B :: \Gamma[k \mapsto_\tau T]_e$

primrec *decE* :: *nat* ⇒ *nat* ⇒ *env* ⇒ *env* (‹$\downarrow_e$›)
where
$\downarrow_e 0\ k\ \Gamma = \Gamma$
$|\ \downarrow_e (Suc\ n)\ k\ \Gamma = \downarrow_e n\ k\ (\Gamma[k \mapsto_\tau Top]_e)$

Note that in a context of the form $B :: \Gamma$, all variables in B with indices smaller than the length of Γ refer to entries in Γ and therefore must not be affected by substitution and lifting. This is the reason why an additional offset $\|\Gamma\|$ needs to be added to the index k in the second clauses of the above functions. Some standard properties of lifting and substitution, which can be proved by structural induction on terms and types, are proved below. Properties of this kind are quite standard for encodings using de Bruijn indices and can also be found in papers by Barras and Werner [2] and Nipkow [3].

lemma *liftE-length* [*simp*]: $\|\uparrow_e n\ k\ \Gamma\| = \|\Gamma\|$
by (*induct* Γ) *simp-all*

lemma *substE-length* [*simp*]: $\|\Gamma[k \mapsto_\tau U]_e\| = \|\Gamma\|$
by (*induct* Γ) *simp-all*

lemma *liftE-nth* [*simp*]:

$(\uparrow_e\ n\ k\ \Gamma)\langle i\rangle = \textit{map-option}\ (\textit{mapB}\ (\uparrow_\tau\ n\ (k + \|\Gamma\| - i - 1)))\ (\Gamma\langle i\rangle)$
proof (*induct* Γ *arbitrary*: *i*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a* Γ)
then show *?case*
by (*cases i*) *auto*
qed

lemma *substE-nth* [*simp*]:
$(\Gamma[0 \mapsto_\tau T]_e)\langle i\rangle = \textit{map-option}\ (\textit{mapB}\ (\lambda U.\ U[\|\Gamma\| - i - 1 \mapsto_\tau T]_\tau))\ (\Gamma\langle i\rangle)$
proof (*induct* Γ *arbitrary*: *i*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a* Γ)
then show *?case*
by (*cases i*) *auto*
qed

lemma *liftT-liftT* [*simp*]:
$i \le j \Longrightarrow j \le i + m \Longrightarrow \uparrow_\tau n\ j\ (\uparrow_\tau m\ i\ T) = \uparrow_\tau (m + n)\ i\ T$
by (*induct T arbitrary*: *i j m n*) *simp-all*

lemma *liftT-liftT'* [*simp*]:
$i + m \le j \Longrightarrow \uparrow_\tau n\ j\ (\uparrow_\tau m\ i\ T) = \uparrow_\tau m\ i\ (\uparrow_\tau n\ (j - m)\ T)$
proof (*induct T arbitrary*: *i j m n*)
case (*TyAll T1 T2*)
then have $\textit{Suc}\ j - m = \textit{Suc}\ (j - m)$
by *arith*
with *TyAll* **show** *?case*
by *simp*
qed *auto*

lemma *lift-size* [*simp*]: $\textit{size}\ (\uparrow_\tau n\ k\ T) = \textit{size}\ T$
by (*induct T arbitrary*: *k*) *simp-all*

lemma *liftT0* [*simp*]: $\uparrow_\tau 0\ i\ T = T$
by (*induct T arbitrary*: *i*) *simp-all*

lemma *lift0* [*simp*]: $\uparrow 0\ i\ t = t$
by (*induct t arbitrary*: *i*) *simp-all*

theorem *substT-liftT* [*simp*]:
$k \le k' \Longrightarrow k' < k + n \Longrightarrow (\uparrow_\tau n\ k\ T)[k' \mapsto_\tau U]_\tau = \uparrow_\tau (n - 1)\ k\ T$
by (*induct T arbitrary*: *k k'*) *simp-all*

theorem *liftT-substT* [*simp*]:
$k \leq k' \Longrightarrow \uparrow_\tau\ n\ k\ (T[k' \mapsto_\tau\ U]_\tau) = \uparrow_\tau\ n\ k\ T[k' + n \mapsto_\tau\ U]_\tau$
by (*induct T arbitrary*: *k k'*) *auto*

theorem *liftT-substT'* [*simp*]: $k' < k \Longrightarrow$
$\uparrow_\tau\ n\ k\ (T[k' \mapsto_\tau\ U]_\tau) = \uparrow_\tau\ n\ (k + 1)\ T[k' \mapsto_\tau \uparrow_\tau\ n\ (k - k')\ U]_\tau$
by (*induct T arbitrary*: *k k'*) *auto*

lemma *liftT-substT-Top* [*simp*]:
$k \leq k' \Longrightarrow \uparrow_\tau\ n\ k'\ (T[k \mapsto_\tau\ Top]_\tau) = \uparrow_\tau\ n\ (Suc\ k')\ T[k \mapsto_\tau\ Top]_\tau$
by (*induct T arbitrary*: *k k'*) *auto*

lemma *liftT-substT-strange*:
$\uparrow_\tau\ n\ k\ T[n + k \mapsto_\tau\ U]_\tau = \uparrow_\tau\ n\ (Suc\ k)\ T[k \mapsto_\tau \uparrow_\tau\ n\ 0\ U]_\tau$
proof (*induct T arbitrary*: *n k*)
case (*TyAll T1 T2*)
then have $\uparrow_\tau\ n\ (Suc\ k)\ T2[Suc\ (n + k) \mapsto_\tau\ U]_\tau = \uparrow_\tau\ n\ (Suc\ (Suc\ k))\ T2[Suc\ k \mapsto_\tau \uparrow_\tau\ n\ 0\ U]_\tau$
by (*metis add-Suc-right*)
with *TyAll* **show** *?case*
by *simp*
qed *auto*

lemma *lift-lift* [*simp*]:
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow\ n'\ k'\ (\uparrow\ n\ k\ t) = \uparrow\ (n + n')\ k\ t$
by (*induct t arbitrary*: *k k'*) *simp-all*

lemma *substT-substT*:
$i \leq j \Longrightarrow T[Suc\ j \mapsto_\tau\ V]_\tau[i \mapsto_\tau\ U[j - i \mapsto_\tau\ V]_\tau]_\tau = T[i \mapsto_\tau\ U]_\tau[j \mapsto_\tau\ V]_\tau$
proof (*induct T arbitrary*: *i j U V*)
case (*TyAll T1 T2*)
then have $T2[Suc\ (Suc\ j) \mapsto_\tau\ V]_\tau[Suc\ i \mapsto_\tau\ U[j - i \mapsto_\tau\ V]_\tau]_\tau =$
$T2[Suc\ i \mapsto_\tau\ U]_\tau[Suc\ j \mapsto_\tau\ V]_\tau$
by (*metis Suc-le-mono diff-Suc-Suc*)
with *TyAll* **show** *?case*
by *auto*
qed *auto*

2.3 Well-formedness

The subtyping and typing judgements to be defined in §2.4 and §2.5 may only operate on types and contexts that are well-formed. Intuitively, a type T is well-formed with respect to a context Γ, if all variables occurring in it are defined in Γ. More precisely, if T contains a type variable *TVar* i, then the ith element of Γ must exist and have the form *TVarB* U.

inductive
well-formed :: *env* $\Rightarrow$ *type* $\Rightarrow$ *bool* (‹- $\vdash_{wf}$ -› [*50*, *50*] *50*)

where
$\textit{wf-TVar}$: $\Gamma\langle i\rangle = \lfloor \textit{TVarB}\ T \rfloor \Longrightarrow \Gamma \vdash_{wf}\ \textit{TVar}\ i$
| $\textit{wf-Top}$: $\Gamma \vdash_{wf}\ \textit{Top}$
| $\textit{wf-arrow}$: $\Gamma \vdash_{wf}\ T \Longrightarrow \Gamma \vdash_{wf}\ U \Longrightarrow \Gamma \vdash_{wf}\ T \rightarrow U$
| $\textit{wf-all}$: $\Gamma \vdash_{wf}\ T \Longrightarrow \textit{TVarB}\ T :: \Gamma \vdash_{wf}\ U \Longrightarrow \Gamma \vdash_{wf}\ (\forall <: T.\ U)$

A context Γ is well-formed, if all types occurring in it only refer to type variables declared "further to the right":

inductive
$\textit{well-formedE} :: \textit{env} \Rightarrow \textit{bool}$ (‹- $\vdash_{wf}$› [50] 50)
and $\textit{well-formedB} :: \textit{env} \Rightarrow \textit{binding} \Rightarrow \textit{bool}$ (‹- $\vdash_{wfB}$ -› [50, 50] 50)
where
$\Gamma \vdash_{wfB}\ B \equiv \Gamma \vdash_{wf}\ \textit{type-ofB}\ B$
| $\textit{wf-Nil}$: $[]\ \vdash_{wf}$
| $\textit{wf-Cons}$: $\Gamma \vdash_{wfB}\ B \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow B :: \Gamma \vdash_{wf}$

The judgement $\Gamma \vdash_{wfB} B$, which denotes well-formedness of the binding B with respect to context Γ, is just an abbreviation for $\Gamma \vdash_{wf}$ *type-ofB B*. We now present a number of properties of the well-formedness judgements that will be used in the proofs in the following sections.

inductive-cases *well-formed-cases*:
$\Gamma \vdash_{wf}\ \textit{TVar}\ i$
$\Gamma \vdash_{wf}\ \textit{Top}$
$\Gamma \vdash_{wf}\ T \rightarrow U$
$\Gamma \vdash_{wf}\ (\forall <: T.\ U)$

inductive-cases *well-formedE-cases*:
$B :: \Gamma \vdash_{wf}$

lemma *wf-TVarB*: $\Gamma \vdash_{wf}\ T \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow \textit{TVarB}\ T :: \Gamma \vdash_{wf}$
by (*rule wf-Cons*) *simp-all*

lemma *wf-VarB*: $\Gamma \vdash_{wf}\ T \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow \textit{VarB}\ T :: \Gamma \vdash_{wf}$
by (*rule wf-Cons*) *simp-all*

lemma *map-is-TVarb*:
$\textit{map is-TVarB}\ \Gamma' = \textit{map is-TVarB}\ \Gamma \Longrightarrow$
$\Gamma\langle i\rangle = \lfloor \textit{TVarB}\ T \rfloor \Longrightarrow \exists\, T.\ \Gamma'\langle i\rangle = \lfloor \textit{TVarB}\ T \rfloor$
proof (*induct* Γ *arbitrary*: Γ' T i)
case *Nil*
then show *?case*
by *auto*
next
case (*Cons a* Γ)
obtain z Γ'' **where** $\Gamma' = z :: \Gamma''$
using *Cons.prems*(*1*) **by** *auto*
with *Cons* **show** *?case*
by (*cases z*) (*auto split*: *nat.splits*)
qed

A type that is well-formed in a context Γ is also well-formed in another context Γ' that contains type variable bindings at the same positions as Γ:

lemma *wf-equallength*:
  assumes *H*: $\Gamma \vdash_{wf} T$
  shows *map is-TVarB* Γ' = *map is-TVarB* $\Gamma \Longrightarrow \Gamma' \vdash_{wf} T$ **using** *H*
  by (*induct arbitrary*: Γ') (*auto intro*: *well-formed.intros dest*: *map-is-TVarb*)

A well-formed context of the form Δ @ B :: Γ remains well-formed if we replace the binding B by another well-formed binding B':

lemma *wfE-replace*:
  Δ @ B :: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wfB} B' \Longrightarrow$ *is-TVarB* B' = *is-TVarB* $B \Longrightarrow$
    Δ @ B' :: $\Gamma \vdash_{wf}$
proof (*induct* Δ)
  case *Nil*
  then show *?case*
    by (*metis append-Nil well-formedE-cases wf-Cons*)
next
  case (*Cons a* Δ)
  then show *?case*
    using *wf-Cons wf-equallength* **by** (*auto elim*!: *well-formedE-cases*)
qed

The following weakening lemmas can easily be proved by structural induction on types and contexts:

lemma *wf-weaken*:
  assumes *H*: Δ @ $\Gamma \vdash_{wf} T$
  shows $\uparrow_e$ (*Suc 0*) *0* Δ @ B :: $\Gamma \vdash_{wf} \uparrow_\tau$ (*Suc 0*) $\|\Delta\|$ T
  using *H*
proof (*induct* Δ @ Γ T *arbitrary*: Δ)
  case *tv*: (*wf-TVar i T*)
  show *?case*
  proof (*cases* $i < \|\Delta\|$)
    case *True*
    with *tv* **show** *?thesis*
      by (*simp add*: *wf-TVar*)
  next
    case *False*
    then have *Suc* $i - \|\Delta\|$ = *Suc* $(i - \|\Delta\|)$
      using *Suc-diff-le linorder-not-less* **by** *blast*
    with *tv False* **show** *?thesis*
      by (*simp add*: *wf-TVar*)
  qed
next
  case *wf-Top*
  then show *?case*
    using *well-formed.wf-Top* **by** *auto*
next
  case (*wf-arrow T U*)

then show *?case*
  by (*simp add*: *well-formed.wf-arrow*)
next
  case (*wf-all T U*)
  then show *?case*
    using *well-formed.wf-all* **by** *force*
qed

lemma *wf-weaken'*: $\Gamma \vdash_{wf} T \Longrightarrow \Delta @ \Gamma \vdash_{wf} \uparrow_\tau \|\Delta\| \ 0 \ T$
proof (*induct* Δ)
  case *Nil*
  then show *?case*
by *auto*
next
  case (*Cons a* Δ)
  then show *?case*
    by (*metis liftT-liftT add-0-right wf-weaken liftE.simps append-Cons*
      append-Nil le-add1 list.size(*3*,*4*))
qed

lemma *wfE-weaken*: $\Delta @ \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wfB} B \Longrightarrow \uparrow_e (Suc\ 0)\ 0\ \Delta @ B :: \Gamma \vdash_{wf}$
proof (*induct* Δ)
  case *Nil*
  then show *?case*
    by (*simp add*: *wf-Cons*)
next
  case (*Cons a* Δ)
  then have $\uparrow_e (Suc\ 0)\ 0\ \Delta @ B :: \Gamma \vdash_{wfB} mapB\ (\uparrow_\tau (Suc\ 0)\ \|\Delta\|)\ a$
    by (*cases a*) (*use wf-weaken* **in** ‹*auto elim*!: *well-formedE-cases*›)
  with *Cons* **show** *?case*
    using *well-formedE-cases wf-Cons* **by** *auto*
qed

Intuitively, lemma *wf-weaken* states that a type T which is well-formed in a context is still well-formed in a larger context, whereas lemma *wfE-weaken* states that a well-formed context remains well-formed when extended with a well-formed binding. Owing to the encoding of variables using de Bruijn indices, the statements of the above lemmas involve additional lifting functions. The typing judgement, which will be described in §2.5, involves the lookup of variables in a context. It has already been pointed out earlier that each entry in a context may only depend on types declared "further to the right". To ensure that a type T stored at position i in an environment Γ is valid in the full environment, as opposed to the smaller environment consisting only of the entries in Γ at positions greater than i, we need to increment the indices of all free type variables in T by *Suc i*:

lemma *wf-liftB*:
  assumes H: $\Gamma \vdash_{wf}$
  shows $\Gamma\langle i\rangle = \lfloor VarB\ T\rfloor \Longrightarrow \Gamma \vdash_{wf} \uparrow_\tau (Suc\ i)\ 0\ T$

using *H*
proof (*induct arbitrary*: *i*)
case *wf-Nil*
then show *?case*
by *auto*
next
case (*wf-Cons* Γ *B*)
then have $\bigwedge j.\ \Gamma\langle j\rangle = \lfloor VarB\ T \rfloor \Longrightarrow B :: \Gamma \vdash_{wf} \uparrow_\tau (Suc\ (Suc\ j))\ 0\ T$
by (*metis Suc-eq-plus1 add-0 append-Nil zero-le liftE.simps*(*1*) *liftT-liftT list.size*(*3*) *wf-weaken*)
with *wf-Cons wf-weaken*[**where** $B = VarB\ T$ **and** $\Delta = []$] **show** *?case*
by (*simp split*: *nat.split-asm*)
qed

We also need lemmas stating that substitution of well-formed types preserves the well-formedness of types and contexts:

theorem *wf-subst*:
$\Delta\ @\ B :: \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Delta[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf} T[\|\Delta\| \mapsto_\tau U]_\tau$
proof (*induct T arbitrary*: Δ)
case (*TVar n* Δ)
then have *1*: $\bigwedge x.\ [\![\Delta\ @\ B :: \Gamma \vdash_{wf} TVar\ x;\ x = \|\Delta\|]\!] \Longrightarrow \Delta[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf} \uparrow_\tau \|\Delta\|\ 0\ U$
by (*metis substE-length wf-weaken'*)
have *2*: $\bigwedge m.\ n - \|\Delta\| = Suc\ m \Longrightarrow n - Suc\ \|\Delta\| = m$
by (*metis Suc-diff-Suc nat.inject zero-less-Suc zero-less-diff*)
show *?case*
using *TVar*
by (*auto simp*: *wf-TVar 1 2 elim*!: *well-formed-cases split*: *nat.split-asm*)
next
case *Top*
then show *?case*
using *wf-Top* **by** *auto*
next
case (*Fun T1 T2*)
then show *?case*
by (*metis substTT.simps*(*3*) *well-formed-cases*(*3*) *wf-arrow*)
next
case (*TyAll T1 T2*)
then have $(TVarB\ T1 :: \Delta)[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf} T2[\|TVarB\ T1 :: \Delta\| \mapsto_\tau U]_\tau$
by (*metis append-Cons well-formed-cases*(*4*))
with *TyAll wf-all* **show** *?case*
by (*auto elim*!: *well-formed-cases*)
qed

theorem *wfE-subst*: $\Delta\ @\ B :: \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Delta[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf}$
proof (*induct* Δ)
case *Nil*
then show *?case*
by (*auto elim*!: *well-formedE-cases*)

next
  case (*Cons a* Δ)
  show *?case*
  proof (*cases a*)
    case (*VarB x1*)
    with *Cons wf-VarB wf-subst* **show** *?thesis*
      by (*auto elim*!: *well-formedE-cases*)
  next
    case (*TVarB x2*)
    with *Cons wf-TVarB wf-subst* **show** *?thesis*
      by (*auto elim*!: *well-formedE-cases*)
  qed
qed

2.4 Subtyping

We now come to the definition of the subtyping judgement $\Gamma \vdash T <: U$.

inductive
  subtyping :: *env* $\Rightarrow$ *type* $\Rightarrow$ *type* $\Rightarrow$ *bool* (‹- /⊢ - <: -› [*50*, *50*, *50*] *50*)
where
  SA-Top: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} S \Longrightarrow \Gamma \vdash S <: Top$
| *SA-refl-TVar*: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} TVar\ i \Longrightarrow \Gamma \vdash TVar\ i <: TVar\ i$
| *SA-trans-TVar*: $\Gamma\langle i\rangle = \lfloor TVarB\ U\rfloor \Longrightarrow$
    $\Gamma \vdash \uparrow_\tau (Suc\ i)\ 0\ U <: T \Longrightarrow \Gamma \vdash TVar\ i <: T$
| *SA-arrow*: $\Gamma \vdash T_1 <: S_1 \Longrightarrow \Gamma \vdash S_2 <: T_2 \Longrightarrow \Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2$
| *SA-all*: $\Gamma \vdash T_1 <: S_1 \Longrightarrow TVarB\ T_1 :: \Gamma \vdash S_2 <: T_2 \Longrightarrow$
    $\Gamma \vdash (\forall{<:}S_1.\ S_2) <: (\forall{<:}T_1.\ T_2)$

The rules *SA-Top* and *SA-refl-TVar*, which appear at the leaves of the derivation tree for a judgement $\Gamma \vdash T <: U$, contain additional side conditions ensuring the well-formedness of the contexts and types involved. In order for the rule *SA-trans-TVar* to be applicable, the context Γ must be of the form $\Gamma_1\ @\ B :: \Gamma_2$, where Γ_1 has the length *i*. Since the indices of variables in *B* can only refer to variables defined in Γ_2, they have to be incremented by *Suc i* to ensure that they point to the right variables in the larger context Γ.

lemma *wf-subtype-env*:
  assumes *PQ*: $\Gamma \vdash P <: Q$
  shows $\Gamma \vdash_{wf}$ **using** *PQ*
  by *induct assumption+*

lemma *wf-subtype*:
  assumes *PQ*: $\Gamma \vdash P <: Q$
  shows $\Gamma \vdash_{wf} P \wedge \Gamma \vdash_{wf} Q$ **using** *PQ*
  by *induct* (*auto intro*: *well-formed.intros elim*!: *wf-equallength*)

lemma *wf-subtypeE*:
  assumes *H*: $\Gamma \vdash T <: U$

and H': $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf}\ T \Longrightarrow \Gamma \vdash_{wf}\ U \Longrightarrow P$
shows P
using *H H' wf-subtype wf-subtype-env* **by** *blast*

By induction on the derivation of $\Gamma \vdash\ T <:\ U$, it can easily be shown that all types and contexts occurring in a subtyping judgement must be well-formed:

lemma *wf-subtype-conj*:
$\Gamma \vdash\ T <:\ U \Longrightarrow \Gamma \vdash_{wf} \wedge \Gamma \vdash_{wf}\ T \wedge \Gamma \vdash_{wf}\ U$
by (*erule wf-subtypeE*) *iprover*

By induction on types, we can prove that the subtyping relation is reflexive:

lemma *subtype-refl*: — A.1
$\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf}\ T \Longrightarrow \Gamma \vdash\ T <:\ T$
by (*induct T arbitrary*: Γ) (*blast intro*:
subtyping.intros wf-Nil wf-TVarB elim: *well-formed-cases*)+

The weakening lemma for the subtyping relation is proved in two steps: by induction on the derivation of the subtyping relation, we first prove that inserting a single type into the context preserves subtyping:

lemma *subtype-weaken*:
assumes H: $\Delta\ @\ \Gamma \vdash\ P <:\ Q$
and *wf*: $\Gamma \vdash_{wfB}\ B$
shows $\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma \vdash\ \uparrow_\tau\ 1\ \|\Delta\|\ P <:\ \uparrow_\tau\ 1\ \|\Delta\|\ Q$ **using** H
proof (*induct* $\Delta\ @\ \Gamma\ P\ Q$ *arbitrary*: Δ)
case *SA-Top*
with *wf* **show** *?case*
by (*auto intro*: *subtyping.SA-Top wfE-weaken wf-weaken*)
next
case *SA-refl-TVar*
with *wf* **show** *?case*
by (*auto intro*!: *subtyping.SA-refl-TVar wfE-weaken dest*: *wf-weaken*)
next
case (*SA-trans-TVar i U T*)
thus *?case*
proof (*cases* $i < \|\Delta\|$)
case *True*
with *SA-trans-TVar*
have $(\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma)\langle i\rangle = \lfloor TVarB\ (\uparrow_\tau\ 1\ (\|\Delta\| - Suc\ i)\ U)\rfloor$
by *simp*
moreover from *True SA-trans-TVar*
have $\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma \vdash$
$\uparrow_\tau\ (Suc\ i)\ 0\ (\uparrow_\tau\ 1\ (\|\Delta\| - Suc\ i)\ U) <:\ \uparrow_\tau\ 1\ \|\Delta\|\ T$
by *simp*
ultimately have $\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma \vdash\ TVar\ i <:\ \uparrow_\tau\ 1\ \|\Delta\|\ T$
by (*rule subtyping.SA-trans-TVar*)
with *True* **show** *?thesis* **by** *simp*
next
case *False*

then have $Suc\ i - \|\Delta\| = Suc\ (i - \|\Delta\|)$ **by** *arith*
with *False SA-trans-TVar* **have** $(\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma)\langle Suc\ i\rangle = \lfloor TVarB\ U\rfloor$
by *simp*
moreover from *False SA-trans-TVar*
have $\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma \vdash \uparrow_\tau\ (Suc\ (Suc\ i))\ 0\ U <: \uparrow_\tau\ 1\ \|\Delta\|\ T$
by *simp*
ultimately have $\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma \vdash TVar\ (Suc\ i) <: \uparrow_\tau\ 1\ \|\Delta\|\ T$
by (*rule subtyping.SA-trans-TVar*)
with *False* **show** *?thesis* **by** *simp*
qed
next
case *SA-arrow*
thus *?case* **by** *simp* (*iprover intro*: *subtyping.SA-arrow*)
next
case $(SA\text{-}all\ T_1\ S_1\ S_2\ T_2\ \Delta)$
with *SA-all(4)* [*of TVarB* $T_1 :: \Delta$]
show *?case* **by** *simp* (*iprover intro*: *subtyping.SA-all*)
qed

All cases are trivial, except for the *SA-trans-TVar* case, which requires a case distinction on whether the index of the variable is smaller than $\|\Delta\|$. The stronger result that appending a new context Δ to a context Γ preserves subtyping can be proved by induction on Δ, using the previous result in the induction step:

lemma *subtype-weaken'*: — A.2
$\Gamma \vdash P <: Q \Longrightarrow \Delta\ @\ \Gamma \vdash_{wf} \Longrightarrow \Delta\ @\ \Gamma \vdash \uparrow_\tau\ \|\Delta\|\ 0\ P <: \uparrow_\tau\ \|\Delta\|\ 0\ Q$
proof (*induct* Δ)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a* Δ)
then have $a :: \Delta\ @\ \Gamma \vdash \uparrow_\tau\ 1\ 0\ (\uparrow_\tau\ \|\Delta\|\ 0\ P) <: \uparrow_\tau\ 1\ 0\ (\uparrow_\tau\ \|\Delta\|\ 0\ Q)$
using *subtype-weaken*[*of* [] $\Delta\ @\ \Gamma$, **where** *B=a*] *liftT-liftT*
by (*fastforce elim*!: *well-formedE-cases*)
then show *?case*
by (*auto elim*!: *well-formedE-cases*)
qed

An unrestricted transitivity rule has the disadvantage that it can be applied in any situation. In order to make the above definition of the subtyping relation *syntax-directed*, the transitivity rule *SA-trans-TVar* is restricted to the case where the type on the left-hand side of the <: operator is a variable. However, the unrestricted transitivity rule can be derived from this definition. In order for the proof to go through, we have to simultaneously prove another property called *narrowing*. The two properties are proved by nested induction. The outer induction is on the size of the type Q, whereas the two inner inductions for proving transitivity and narrowing are on the derivation of the subtyping judgements. The transitivity property is needed in the proof of narrowing, which is by induction on the derivation of Δ @ $TVarB\ Q$ :: $\Gamma \vdash M <: N$. In the case corresponding to the rule *SA-trans-TVar*, we must prove Δ @ $TVarB\ P$ :: $\Gamma \vdash TVar\ i <: T$. The only interesting case is the one where $i = \|\Delta\|$. By induction hypothesis, we know that Δ @ $TVarB\ P$ :: $\Gamma \vdash \uparrow_\tau (i + 1)\ 0\ Q <: T$ and (Δ @ $TVarB\ Q$ :: $\Gamma)\langle i \rangle = \lfloor TVarB\ Q \rfloor$. By assumption, we have $\Gamma \vdash P <: Q$ and hence Δ @ $TVarB\ P$ :: $\Gamma \vdash \uparrow_\tau (i + 1)\ 0\ P <: \uparrow_\tau (i + 1)\ 0\ Q$ by weakening. Since $\uparrow_\tau (i + 1)\ 0\ Q$ has the same size as Q, we can use the transitivity property, which yields Δ @ $TVarB\ P$ :: $\Gamma \vdash \uparrow_\tau (i + 1)\ 0\ P <: T$. The claim then follows easily by an application of *SA-trans-TVar*.

lemma *subtype-trans*: — A.3
  $\Gamma \vdash S <: Q \Longrightarrow \Gamma \vdash Q <: T \Longrightarrow \Gamma \vdash S <: T$
  Δ @ $TVarB\ Q$ :: $\Gamma \vdash M <: N \Longrightarrow \Gamma \vdash P <: Q \Longrightarrow$
    Δ @ $TVarB\ P$ :: $\Gamma \vdash M <: N$
  using *wf-measure-size*
proof (*induct Q arbitrary*: $\Gamma\ S\ T\ \Delta\ P\ M\ N$ *rule*: *wf-induct-rule*)
  case (*less Q*)
  have *tr*: $\Gamma \vdash Q' <: T \Longrightarrow size\ Q = size\ Q' \Longrightarrow \Gamma \vdash S <: T$
    if $\Gamma \vdash S <: Q'$ **for** $\Gamma\ S\ T\ Q'$
    using *that*
  proof (*induct arbitrary*: *T*)
    case *SA-Top*
    from *SA-Top*(*3*) **show** *?case*
      by *cases* (*auto intro*: *subtyping.SA-Top SA-Top*)
  next
    case *SA-refl-TVar* **show** *?case* **by** *fact*
  next
    case *SA-trans-TVar*
    thus *?case* **by** (*auto intro*: *subtyping.SA-trans-TVar*)
  next
    case (*SA-arrow* $\Gamma\ T_1\ S_1\ S_2\ T_2$)
    note *SA-arrow′* = *SA-arrow*
    from *SA-arrow*(*5*) **show** *?case*
    proof *cases*
      case *SA-Top*
      with *SA-arrow* **show** *?thesis*
        by (*auto intro*: *subtyping.SA-Top wf-arrow elim*: *wf-subtypeE*)
    next
      case (*SA-arrow* $T_1{}'\ T_2{}'$)
      from *SA-arrow SA-arrow′* **have** $\Gamma \vdash S_1 \to S_2 <: T_1{}' \to T_2{}'$
        by (*auto intro*!: *subtyping.SA-arrow intro*: *less*(*1*) [*of* T_1] *less*(*1*) [*of* T_2])
      with *SA-arrow* **show** *?thesis* **by** *simp*
    qed
  next
    case (*SA-all* $\Gamma\ T_1\ S_1\ S_2\ T_2$)
    note *SA-all′* = *SA-all*
    from *SA-all*(*5*) **show** *?case*

proof *cases*
  case *SA-Top*
  with *SA-all* **show** *?thesis* **by** (*auto intro*!:
      subtyping.SA-Top wf-all intro: *wf-equallength elim*: *wf-subtypeE*)
next
  case (*SA-all* $T_1{}'$ $T_2{}'$)
  from *SA-all SA-all'* **have** $\Gamma \vdash T_1{}' <: S_1$
    by − (*rule less*(*1*), *simp-all*)
  moreover from *SA-all SA-all'* **have** *TVarB* $T_1{}'$ :: $\Gamma \vdash S_2 <: T_2$
    by − (*rule less*(*2*) [*of* - [], *simplified*], *simp-all*)
  with *SA-all SA-all'* **have** *TVarB* $T_1{}'$ :: $\Gamma \vdash S_2 <: T_2{}'$
    by − (*rule less*(*1*), *simp-all*)
  ultimately have $\Gamma \vdash (\forall <:S_1.\ S_2) <: (\forall <:T_1{}'.\ T_2{}')$
    by (*rule subtyping.SA-all*)
  with *SA-all* **show** *?thesis* **by** *simp*
qed
qed
{
  case *1*
  thus *?case* **using** *refl* **by** (*rule tr*)
next
  case *2*
  from *2*(*1*) **show** Δ @ *TVarB P* :: $\Gamma \vdash M <: N$
  proof (*induct* Δ @ *TVarB Q* :: Γ *M N arbitrary*: Δ)
    case *SA-Top*
    with *2* **show** *?case* **by** (*auto intro*!: *subtyping.SA-Top*
      intro: *wf-equallength wfE-replace elim*!: *wf-subtypeE*)
  next
    case *SA-refl-TVar*
    with *2* **show** *?case* **by** (*auto intro*!: *subtyping.SA-refl-TVar*
      intro: *wf-equallength wfE-replace elim*!: *wf-subtypeE*)
  next
    case (*SA-trans-TVar i U T*)
    show *?case*
    proof (*cases* $i < \|\Delta\|$)
      case *True*
      with *SA-trans-TVar* **show** *?thesis*
        by (*auto intro*!: *subtyping.SA-trans-TVar*)
    next
      case *False*
      note *False'* = *False*
      show *?thesis*
      proof (*cases* $i = \|\Delta\|$)
        case *True*
        from *SA-trans-TVar* **have** (Δ @ [*TVarB P*]) @ $\Gamma \vdash_{wf}$
          by (*auto elim*!: *wf-subtypeE*)
        with ‹$\Gamma \vdash P <: Q$›
        have (Δ @ [*TVarB P*]) @ $\Gamma \vdash \uparrow_\tau$ $\|\Delta$ @ [*TVarB P*]$\|$ *0 P* <: $\uparrow_\tau$ $\|\Delta$ @
[*TVarB P*]$\|$ *0 Q*

by (*rule subtype-weaken′*)
with *SA-trans-TVar True False* **have** Δ @ *TVarB P* :: $\Gamma \vdash \uparrow_\tau$ (*Suc* $\|\Delta\|$) *0 P* <: *T*
by − (*rule tr*, *simp*+)
with *True* **and** *False* **and** *SA-trans-TVar* **show** *?thesis*
by (*auto intro*!: *subtyping.SA-trans-TVar*)
next
case *False*
with *False′* **have** $i - \|\Delta\| = Suc\ (i - \|\Delta\| - 1)$ **by** *arith*
with *False False′ SA-trans-TVar* **show** *?thesis*
by (*simp add*: *subtyping.SA-trans-TVar*)
qed
qed
next
case *SA-arrow*
thus *?case* **by** (*auto intro*!: *subtyping.SA-arrow*)
next
case (*SA-all* T_1 S_1 S_2 T_2)
thus *?case*
using *subtyping.SA-all* **by** *auto*
qed
}
qed

In the proof of the preservation theorem presented in §2.6, we will also need a substitution theorem, which is proved by induction on the subtyping derivation:

lemma *substT-subtype*: — A.10
assumes *H*: Δ @ *TVarB Q* :: $\Gamma \vdash S$ <: *T*
shows $\Gamma \vdash P$ <: $Q \Longrightarrow \Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash S[\|\Delta\| \mapsto_\tau P]_\tau$ <: $T[\|\Delta\| \mapsto_\tau P]_\tau$
using *H*
proof (*induct* Δ @ *TVarB Q* :: Γ *S T arbitrary*: Δ)
case (*SA-Top S*)
then show *?case*
by (*simp add*: *subtyping.SA-Top wfE-subst wf-subst wf-subtype*)
next
case (*SA-refl-TVar i*)
then show *?case*
using *subtype-refl wfE-subst wf-subst wf-subtype* **by** *presburger*
next
case §: (*SA-trans-TVar i U T*)
show *?case*
proof −
have $\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash \uparrow_\tau \|\Delta\|$ *0 P* <: $T[\|\Delta\| \mapsto_\tau P]_\tau$
if $i = \|\Delta\|$
using *that* §
by *simp* (*smt* (*verit*, *best*) *substE-length subtype-trans*(*1*) *subtype-weaken′ wf-subtype-env*)
moreover have $\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash$ *TVar* $(i - Suc\ 0)$ <: $T[\|\Delta\| \mapsto_\tau P]_\tau$

```
      if ‖Δ‖ < i
    proof (cases i − ‖Δ‖)
      case 0
      with that show ?thesis
        by linarith
    next
      case (Suc n)
      then have i − Suc ‖Δ‖ = n
        by simp
      with § SA-trans-TVar Suc show ?thesis by simp
    qed
    moreover have Δ[0 ↦τ P]e @ Γ ⊢ TVar i <: T[‖Δ‖ ↦τ P]τ
      if i < ‖Δ‖
    proof −
      have Suc (‖Δ‖ − Suc 0) = ‖Δ‖
        using that by linarith
      with that § show ?thesis
        by (simp add: SA-trans-TVar split: nat.split-asm)
    qed
    ultimately show ?thesis
      by auto
  qed
next
  case (SA-arrow T1 S1 S2 T2)
  then show ?case
    by (simp add: subtyping.SA-arrow)
next
  case §: (SA-all T1 S1 S2 T2)
  then show ?case
    by (simp add: SA-all)
qed

lemma subst-subtype:
  assumes H: Δ @ VarB V :: Γ ⊢ T <: U
  shows ↓e 1 0 Δ @ Γ ⊢ ↓τ 1 ‖Δ‖ T <: ↓τ 1 ‖Δ‖ U
  using H
proof (induct Δ @ VarB V :: Γ T U arbitrary: Δ)
  case (SA-Top S)
  then show ?case
    by (simp add: subtyping.SA-Top wfE-subst wf-Top wf-subst)
next
  case (SA-refl-TVar i)
  then show ?case
    by (metis One-nat-def decE.simps decT.simps subtype-refl wfE-subst wf-Top
        wf-subst)
next
  case §: (SA-trans-TVar i U T)
  show ?case
  proof −
```

have $\Delta[\mathit{0} \mapsto_\tau \mathit{Top}]_e \mathbin{@} \Gamma \vdash \mathit{Top} <: T[\|\Delta\| \mapsto_\tau \mathit{Top}]_\tau$ **if** $i = \|\Delta\|$
using *that* § **by** (*simp split*: *nat.split-asm*)
moreover have $\Delta[\mathit{0} \mapsto_\tau \mathit{Top}]_e \mathbin{@} \Gamma \vdash \mathit{TVar}\ (i - \mathit{Suc}\ \mathit{0}) <: T[\|\Delta\| \mapsto_\tau \mathit{Top}]_\tau$
if $\|\Delta\| < i$
proof (*cases* $i - \|\Delta\|$)
case *0*
with *that* **show** *?thesis*
by *linarith*
next
case (*Suc n*)
then have $i - \mathit{Suc}\ \|\Delta\| = n$
by *simp*
with § *SA-trans-TVar Suc* **show** *?thesis* **by** *simp*
qed
moreover have $\Delta[\mathit{0} \mapsto_\tau \mathit{Top}]_e \mathbin{@} \Gamma \vdash \mathit{TVar}\ i <: T[\|\Delta\| \mapsto_\tau \mathit{Top}]_\tau$
if $\|\Delta\| > i$
proof –
have $\mathit{Suc}\ (\|\Delta\| - \mathit{Suc}\ \mathit{0}) = \|\Delta\|$
using *that* **by** *linarith*
with *that* § **show** *?thesis*
by (*simp add*: *SA-trans-TVar split*: *nat.split-asm*)
qed
ultimately show *?thesis*
by *auto*
qed
next
case (*SA-arrow* $T_1\ S_1\ S_2\ T_2$)
then show *?case*
by (*simp add*: *subtyping.SA-arrow*)
next
case §: (*SA-all* $T_1\ S_1\ S_2\ T_2$)
then show *?case*
by (*simp add*: *SA-all*)
qed

2.5 Typing

We are now ready to give a definition of the typing judgement $\Gamma \vdash t : T$.

inductive
typing :: *env* ⇒ *trm* ⇒ *type* ⇒ *bool* (‹- /⊢ - : -› [*50*, *50*, *50*] *50*)
where
T-Var: $\Gamma \vdash_{wf} \Longrightarrow \Gamma\langle i\rangle = \lfloor \mathit{VarB}\ U \rfloor \Longrightarrow T = \uparrow_\tau (\mathit{Suc}\ i)\ \mathit{0}\ U \Longrightarrow \Gamma \vdash \mathit{Var}\ i : T$
| *T-Abs*: $\mathit{VarB}\ T_1 :: \Gamma \vdash t_2 : T_2 \Longrightarrow \Gamma \vdash (\lambda{:}T_1.\ t_2) : T_1 \to \downarrow_\tau \mathit{1}\ \mathit{0}\ T_2$
| *T-App*: $\Gamma \vdash t_1 : T_{11} \to T_{12} \Longrightarrow \Gamma \vdash t_2 : T_{11} \Longrightarrow \Gamma \vdash t_1 \cdot t_2 : T_{12}$
| *T-TAbs*: $\mathit{TVarB}\ T_1 :: \Gamma \vdash t_2 : T_2 \Longrightarrow \Gamma \vdash (\lambda{<:}T_1.\ t_2) : (\forall{<:}T_1.\ T_2)$
| *T-TApp*: $\Gamma \vdash t_1 : (\forall{<:}T_{11}.\ T_{12}) \Longrightarrow \Gamma \vdash T_2 <: T_{11} \Longrightarrow$
$\Gamma \vdash t_1 \cdot_\tau T_2 : T_{12}[\mathit{0} \mapsto_\tau T_2]_\tau$
| *T-Sub*: $\Gamma \vdash t : S \Longrightarrow \Gamma \vdash S <: T \Longrightarrow \Gamma \vdash t : T$

Note that in the rule *T-Var*, the indices of the type U looked up in the context Γ need to be incremented in order for the type to be well-formed with respect to Γ. In the rule *T-Abs*, the type T_2 of the abstraction body t_2 may not contain the variable with index *0*, since it is a term variable. To compensate for the disappearance of the context element *VarB* T_1 in the conclusion of thy typing rule, the indices of all free type variables in T_2 have to be decremented by *1*.

theorem *wf-typeE1*:
  assumes *H*: $\Gamma \vdash t : T$
  shows $\Gamma \vdash_{wf}$ **using** *H*
  by *induct* (*blast elim*: *well-formedE-cases*)+

theorem *wf-typeE2*:
  assumes *H*: $\Gamma \vdash t : T$
  shows $\Gamma \vdash_{wf} T$ **using** *H*
proof *induct*
  case (*T-Var* Γ *i U T*)
  then show *?case*
    by (*simp add*: *wf-liftB*)
next
  case (*T-Abs* T_1 Γ t_2 T_2)
  then have $\Gamma \vdash_{wf} T_2[0 \mapsto_\tau Top]_\tau$
    by (*metis append-Nil list.size*(*3*) *substE.simps*(*1*) *wf-Top wf-subst*)
  with *T-Abs wf-arrow wf-typeE1* **show** *?case*
    by (*metis One-nat-def decT.simps*(*1*,*2*) *type-ofB.simps*(*1*) *well-formedE-cases*)
next
  case (*T-App* Γ t_1 T_{11} T_{12} t_2)
  then show *?case*
    using *well-formed-cases*(*3*) **by** *blast*
next
  case (*T-TAbs* T_1 Γ t_2 T_2)
  then show *?case*
    by (*metis type-ofB.simps*(*2*) *well-formedE-cases wf-all wf-typeE1*)
next
  case (*T-TApp* Γ t_1 T_{11} T_{12} T_2)
  then show *?case*
    by (*metis append-Nil list.size*(*3*) *substE.simps*(*1*) *well-formed-cases*(*4*) *wf-subst wf-subtype*)
next
  case (*T-Sub* Γ *t S T*)
  then show *?case*
    by (*auto elim*: *wf-subtypeE*)
qed

Like for the subtyping judgement, we can again prove that all types and contexts involved in a typing judgement are well-formed:

lemma *wf-type-conj*: $\Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wf} \wedge \Gamma \vdash_{wf} T$
  by (*frule wf-typeE1*, *drule wf-typeE2*) *iprover*

The narrowing theorem for the typing judgement states that replacing the type of a variable in the context by a subtype preserves typability:

```
lemma narrow-type: — A.7
  assumes H: Δ @ TVarB Q :: Γ ⊢ t : T
  shows Γ ⊢ P <: Q ⟹ Δ @ TVarB P :: Γ ⊢ t : T
  using H
proof (induct Δ @ TVarB Q :: Γ t T arbitrary: Δ)
  case §: (T-Var i U T)
  show ?case
  proof (intro T-Var)
    show Δ @ TVarB P :: Γ ⊢wf
      using §
      by (metis is-TVarB.simps(2) type-ofB.simps(2) wfE-replace wf-subtypeE)
  next
    show (Δ @ TVarB P :: Γ)⟨i⟩ = ⌊VarB U⌋
    proof (cases i < ‖Δ‖)
      case True
      with § show ?thesis
        by simp
    next
      case False
      with § show ?thesis
        by (simp split: nat.splits)
    qed
  next
    show T = ↑τ (Suc i) 0 U
      by (simp add: §.hyps)
  qed
next
  case (T-Abs T1 t2 T2)
  then show ?case
    using typing.T-Abs by auto
next
  case (T-TApp t1 T11 T12 T2)
  then show ?case
    using subtype-trans(2) typing.T-TApp by blast
next
  case (T-Sub t S T)
  then show ?case
    using subtype-trans(2) typing.T-Sub by blast
qed (auto intro: typing.intros)

lemma subtype-refl':
  assumes t: Γ ⊢ t : T
  shows Γ ⊢ T <: T
  using subtype-refl t wf-typeE1 wf-typeE2 by blast

lemma Abs-type: — A.13(1)
  assumes H: Γ ⊢ (λ:S. s) : T
```

```
  shows Γ ⊢ T <: U → U′ ⟹
    (⋀S′. Γ ⊢ U <: S ⟹ VarB S :: Γ ⊢ s : S′ ⟹
      Γ ⊢ ↓τ 1 0 S′ <: U′ ⟹ P) ⟹ P
  using H
proof (induct Γ λ:S. s T arbitrary: U U′ S s P)
  case (T-Abs T_1 Γ t_2 T_2)
  from ‹Γ ⊢ T_1 → ↓τ 1 0 T_2 <: U → U′›
  obtain ty1: Γ ⊢ U <: T_1 and ty2: Γ ⊢ ↓τ 1 0 T_2 <: U′
    by cases simp-all
  from ty1 ‹VarB T_1 :: Γ ⊢ t_2 : T_2› ty2
  show ?case by (rule T-Abs)
next
  case (T-Sub Γ S′ T)
  from ‹Γ ⊢ S′ <: T› and ‹Γ ⊢ T <: U → U′›
  have Γ ⊢ S′ <: U → U′ by (rule subtype-trans(1))
  then show ?case
    by (rule T-Sub) (rule T-Sub(5))
qed

lemma Abs-type′:
  assumes H: Γ ⊢ (λ:S. s) : U → U′
    and R: ⋀S′. Γ ⊢ U <: S ⟹ VarB S :: Γ ⊢ s : S′ ⟹
    Γ ⊢ ↓τ 1 0 S′ <: U′ ⟹ P
  shows P
  using Abs-type H R subtype-refl′ by blast

lemma TAbs-type: — A.13(2)
  assumes H: Γ ⊢ (λ<:S. s) : T
  shows Γ ⊢ T <: (∀<:U. U′) ⟹
    (⋀S′. Γ ⊢ U <: S ⟹ TVarB U :: Γ ⊢ s : S′ ⟹
      TVarB U :: Γ ⊢ S′ <: U′ ⟹ P) ⟹ P
  using H
proof (induct Γ λ<:S. s T arbitrary: U U′ S s P)
  case (T-TAbs T_1 Γ t_2 T_2)
  from ‹Γ ⊢ (∀<:T_1. T_2) <: (∀<:U. U′)›
  obtain ty1: Γ ⊢ U <: T_1 and ty2: TVarB U :: Γ ⊢ T_2 <: U′
    by cases simp-all
  from ‹TVarB T_1 :: Γ ⊢ t_2 : T_2›
  have TVarB U :: Γ ⊢ t_2 : T_2 using ty1
    by (rule narrow-type [of [], simplified])
  with ty1 show ?case using ty2 by (rule T-TAbs)
next
  case (T-Sub Γ S′ T)
  from ‹Γ ⊢ S′ <: T› and ‹Γ ⊢ T <: (∀<:U. U′)›
  have Γ ⊢ S′ <: (∀<:U. U′) by (rule subtype-trans(1))
  then show ?case
    by (rule T-Sub) (rule T-Sub(5))
qed
```

lemma *TAbs-type′*:
  assumes *H*: $\Gamma \vdash (\lambda{<:}S.\ s) : (\forall{<:}U.\ U')$
    and *R*: $\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow TVarB\ U :: \Gamma \vdash s : S' \Longrightarrow$
    $TVarB\ U :: \Gamma \vdash S' <: U' \Longrightarrow P$
  shows *P* **using** *H subtype-refl′* [*OF H*]
  by (*rule TAbs-type*) (*rule R*)

lemma *T-eq*: $\Gamma \vdash t : T \Longrightarrow T = T' \Longrightarrow \Gamma \vdash t : T'$ **by** *simp*

The weakening theorem states that inserting a binding *B* does not affect typing:

lemma *type-weaken*:
  assumes *H*: $\Delta\ @\ \Gamma \vdash t : T$
  shows $\Gamma \vdash_{wfB} B \Longrightarrow$
    $\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma \vdash \uparrow\ 1\ \|\Delta\|\ t : \uparrow_\tau\ 1\ \|\Delta\|\ T$ **using** *H*
proof (*induct* $\Delta\ @\ \Gamma\ t\ T$ *arbitrary*: Δ)
  case §: (*T-Var i U T*)
  show *?case*
  proof −
    have $\uparrow_e$ (*Suc 0*) *0* $\Delta\ @\ B :: \Gamma \vdash$ *Var i* : $\uparrow_\tau$ (*Suc 0*) $\|\Delta\|\ T$
      if $i < \|\Delta\|$
      using § *that*
      by (*intro T-Var*) (*auto simp*: *elim*!: *wfE-weaken*)
    moreover have $\uparrow_e$ (*Suc 0*) *0* $\Delta\ @\ B :: \Gamma \vdash$ *Var* (*Suc i*) : $\uparrow_\tau$ (*Suc 0*) $\|\Delta\|\ T$
      if $\neg\ i < \|\Delta\|$
      using § *that*
      by (*intro T-Var*) (*auto simp*: *Suc-diff-le elim*!: *wfE-weaken*)
    ultimately show *?thesis*
      by *auto*
  qed
next
  case (*T-Abs* $T_1\ t_2\ T_2$)
  then show *?case*
    using *typing.T-Abs* **by** *simp*
next
  case (*T-App* $t_1\ T_{11}\ T_{12}\ t_2$)
  then show *?case*
    using *typing.T-App* **by** *force*
next
  case (*T-TAbs* $T_1\ t_2\ T_2$)
  then show *?case*
    using *typing.T-TAbs* **by** *force*
next
  case §: (*T-TApp* $t_1\ T_{11}\ T_{12}\ T_2$)
  show *?case*
  proof (*cases* Δ)
    case *Nil*
    with § *liftT-substT-strange* [*of - 0*] **show** *?thesis*
     apply *simp*

by (*metis One-nat-def T-TApp append-Nil liftE.simps*(*1*) *list.size*(*3*) *subtype-weaken*)
next
case (*Cons a list*)
with § **show** *?thesis*
by (*metis T-TApp diff-Suc-1′ diff-Suc-Suc length-Cons lift.simps*(*5*) *liftT.simps*(*4*) *liftT-substT′ subtype-weaken zero-less-Suc*)
qed
next
case (*T-Sub t S T*)
with *subtype-weaken typing.T-Sub* **show** *?case*
by *blast*
qed

We can strengthen this result, so as to mean that concatenating a new context Δ to the context Γ preserves typing:

lemma *type-weaken′*: — A.5(6)
$\Gamma \vdash t : T \Longrightarrow \Delta \mathbin{@} \Gamma \vdash_{wf} \Longrightarrow \Delta \mathbin{@} \Gamma \vdash \uparrow \|\Delta\|\ 0\ t : \uparrow_\tau \|\Delta\|\ 0\ T$
proof (*induct* Δ)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a* Δ)
with *type-weaken* [**where** *B*=*a*, *of* []] **show** *?case*
by (*fastforce simp*: *elim*!: *well-formedE-cases*)
qed

This property is proved by structural induction on the context Δ, using the previous result in the induction step. In the proof of the preservation theorem, we will need two substitution theorems for term and type variables, both of which are proved by induction on the typing derivation. Since term and type variables are stored in the same context, we again have to decrement the free type variables in Δ and T by *1* in the substitution rule for term variables in order to compensate for the disappearance of the variable.

theorem *subst-type*: — A.8
assumes *H*: $\Delta \mathbin{@} \mathit{VarB}\ U :: \Gamma \vdash t : T$
shows $\Gamma \vdash u : U \Longrightarrow$
$\downarrow_e 1\ 0\ \Delta \mathbin{@} \Gamma \vdash (\mathit{subst}\ t\ (\mathit{length}\ \Delta)\ u) : \downarrow_\tau 1\ \|\Delta\|\ T$ **using** *H*
proof (*induct* $\Delta \mathbin{@} \mathit{VarB}\ U :: \Gamma\ t\ T$ *arbitrary*: Δ)
case §: (*T-Var i U T*)
show *?case*
proof −
have $\Delta[0 \mapsto_\tau \mathit{Top}]_e \mathbin{@} \Gamma \vdash \uparrow \|\Delta\|\ 0\ u : T[\|\Delta\| \mapsto_\tau \mathit{Top}]_\tau$
if $i = \|\Delta\|$
using § *that type-weaken′ wfE-subst wf-Top* **by** *fastforce*
moreover have $\Delta[0 \mapsto_\tau \mathit{Top}]_e \mathbin{@} \Gamma \vdash \mathit{Var}\ (i - \mathit{Suc}\ 0) : T[\|\Delta\| \mapsto_\tau \mathit{Top}]_\tau$
if $\|\Delta\| < i$

```
      using § that
      by (simp split: nat-diff-split-asm nat.split-asm add: T-Var wfE-subst wf-Top)
    moreover have Δ[0 ↦τ Top]e @ Γ ⊢ Var i : T[‖Δ‖ ↦τ Top]τ
      if ‖Δ‖ > i
    proof −
      have Suc (‖Δ‖ − Suc 0) = ‖Δ‖
        using that by force
      then show ?thesis
        using § that wfE-subst wf-Top by (intro T-Var) auto
  qed
    ultimately show ?thesis
      by auto
  qed
next
  case §: (T-Abs T1 t2 T2)
  then show ?case
    by (simp add: T-Abs [THEN T-eq] substT-substT [symmetric])
next
  case (T-App t1 T11 T12 t2)
  then show ?case
    by (simp add: typing.T-App)
next
  case §: (T-TAbs T1 t2 T2)
  then show ?case
    by (simp add: typing.T-TAbs)
next
  case §: (T-TApp t1 T11 T12 T2)
  then show ?case
      by (auto intro!: T-TApp [THEN T-eq] dest: subst-subtype simp flip: sub-
stT-substT)
next
  case (T-Sub t S T)
  then show ?case
    using subst-subtype typing.T-Sub by blast
qed

theorem substT-type: — A.11
  assumes H: Δ @ TVarB Q :: Γ ⊢ t : T
  shows Γ ⊢ P <: Q ⟹
    Δ[0 ↦τ P]e @ Γ ⊢ t[‖Δ‖ ↦τ P] : T[‖Δ‖ ↦τ P]τ using H
proof (induct Δ @ TVarB Q :: Γ t T arbitrary: Δ)
  case §: (T-Var i U T)
  show ?case
  proof −
    have Δ[0 ↦τ P]e @ Γ ⊢ Var (i − Suc 0) : T[‖Δ‖ ↦τ P]τ
      if ‖Δ‖ < i
      using § that
     by (simp split: nat-diff-split-asm nat.split-asm add: T-Var wfE-subst wf-subtype)
    moreover have Δ[0 ↦τ P]e @ Γ ⊢ Var i : T[‖Δ‖ ↦τ P]τ
```

if $\|\Delta\| = i$
using § *that* **by** (*intro T-Var* [**where** U=($U[\|\Delta\| - Suc\ i \mapsto_\tau P]_\tau$)]) *auto*
moreover have $\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash Var\ i : T[\|\Delta\| \mapsto_\tau P]_\tau$
if $\|\Delta\| > i$
proof −
have $Suc\ (\|\Delta\| - Suc\ 0) = \|\Delta\|$
using *that* **by** *auto*
with § *that* **show** *?thesis*
by (*simp add*: *T-Var wfE-subst wf-subtype*)
qed
ultimately show *?thesis*
by *fastforce*
qed
next
case §: (*T-Abs* T_1 t_2 T_2)
then show *?case*
by (*simp add*: *T-Abs* [*THEN T-eq*] *substT-substT* [*symmetric*])
next
case (*T-TApp* t_1 T_{11} T_{12} T_2)
then show *?case*
using *substT-substT*[*of* 0 $\|\Delta\|$ T_{12} P T_2] *substT-subtype*
typing.T-TApp[*of* - - - $T_{12}[Suc\ \|\Delta\| \mapsto_\tau P]_\tau$ $T_2[\|\Delta\| \mapsto_\tau P]_\tau$]
by *auto*
next
case (*T-Sub t S T*)
then show *?case*
using *substT-subtype typing.T-Sub* **by** *blast*
qed (*auto simp*: *typing.T-App typing.T-TAbs*)

2.6 Evaluation

For the formalization of the evaluation strategy, it is useful to first define a set of *canonical values* that are not evaluated any further. The canonical values of call-by-value $F_{<:}$ are exactly the abstractions over term and type variables:

inductive-set
value :: *trm set*
where
Abs: $(\lambda{:}T.\ t) \in value$
| *TAbs*: $(\lambda{<:}T.\ t) \in value$

The notion of a *value* is now used in the defintion of the evaluation relation $t \longmapsto t'$. There are several ways for defining this evaluation relation: Aydemir et al. [1] advocate the use of *evaluation contexts* that allow to separate the description of the "immediate" reduction rules, i.e. β-reduction, from the description of the context in which these reductions may occur in. The rationale behind this approach is to keep the formalization more modular. We will take a closer look at this style of presentation in section §4. For the rest of this section, we will use a different approach: both the "immediate" reductions and the reduction context are described within the same inductive definition, where the context is described by additional congruence rules.

inductive
  eval :: *trm* ⇒ *trm* ⇒ *bool* (**infixl** ‹⟼› *50*)
where
  E-Abs: $v_2 \in value \Longrightarrow (\lambda{:}T_{11}.\ t_{12}) \cdot v_2 \longmapsto t_{12}[0 \mapsto v_2]$
| *E-TAbs*: $(\lambda{<:}T_{11}.\ t_{12}) \cdot_\tau T_2 \longmapsto t_{12}[0 \mapsto_\tau T_2]$
| *E-App1*: $t \longmapsto t' \Longrightarrow t \cdot u \longmapsto t' \cdot u$
| *E-App2*: $v \in value \Longrightarrow t \longmapsto t' \Longrightarrow v \cdot t \longmapsto v \cdot t'$
| *E-TApp*: $t \longmapsto t' \Longrightarrow t \cdot_\tau T \longmapsto t' \cdot_\tau T$

Here, the rules *E-Abs* and *E-TAbs* describe the "immediate" reductions, whereas *E-App1*, *E-App2*, and *E-TApp* are additional congruence rules describing reductions in a context. The most important theorems of this section are the *preservation* theorem, stating that the reduction of a well-typed term does not change its type, and the *progress* theorem, stating that reduction of a well-typed term does not "get stuck" – in other words, every well-typed, closed term t is either a value, or there is a term t' to which t can be reduced. The preservation theorem is proved by induction on the derivation of $\Gamma \vdash t : T$, followed by a case distinction on the last rule used in the derivation of $t \longmapsto t'$.

theorem *preservation*: — A.20
  assumes H: $\Gamma \vdash t : T$
  shows $t \longmapsto t' \Longrightarrow \Gamma \vdash t' : T$ **using** H
proof (*induct arbitrary*: t')
  case (*T-Var* Γ i U T t')
  from ‹*Var* $i \longmapsto t'$›
  show *?case* **by** *cases*
next
  case (*T-Abs* T_1 Γ t_2 T_2 t')
  from ‹$(\lambda{:}T_1.\ t_2) \longmapsto t'$›
  show *?case* **by** *cases*
next
  case (*T-App* Γ t_1 T_{11} T_{12} t_2 t')
  from ‹$t_1 \cdot t_2 \longmapsto t'$›
  show *?case*
  proof *cases*
    case (*E-Abs* T_{11}' t_{12})
    with *T-App* **have** $\Gamma \vdash (\lambda{:}T_{11}'.\ t_{12}) : T_{11} \to T_{12}$ **by** *simp*
    then obtain S'
      where T_{11}: $\Gamma \vdash T_{11} <: T_{11}'$
      and t_{12}: *VarB* T_{11}' :: $\Gamma \vdash t_{12} : S'$
      and S': $\Gamma \vdash S'[0 \mapsto_\tau Top]_\tau <: T_{12}$ **by** (*rule Abs-type'* [*simplified*]) *blast*
    from ‹$\Gamma \vdash t_2 : T_{11}$›
    have $\Gamma \vdash t_2 : T_{11}'$ **using** T_{11} **by** (*rule T-Sub*)
    with t_{12} **have** $\Gamma \vdash t_{12}[0 \mapsto t_2] : S'[0 \mapsto_\tau Top]_\tau$

by (*rule subst-type* [**where** Δ=[], *simplified*])
hence $\Gamma \vdash t_{12}[0 \mapsto t_2] : T_{12}$ **using** S' **by** (*rule T-Sub*)
with *E-Abs* **show** *?thesis* **by** *simp*
next
case (*E-App1* t'')
from ‹$t_1 \longmapsto t''$›
have $\Gamma \vdash t'' : T_{11} \rightarrow T_{12}$ **by** (*rule T-App*)
hence $\Gamma \vdash t'' \cdot t_2 : T_{12}$ **using** ‹$\Gamma \vdash t_2 : T_{11}$›
by (*rule typing.T-App*)
with *E-App1* **show** *?thesis* **by** *simp*
next
case (*E-App2* t'')
from ‹$t_2 \longmapsto t''$›
have $\Gamma \vdash t'' : T_{11}$ **by** (*rule T-App*)
with *T-App*(*1*) **have** $\Gamma \vdash t_1 \cdot t'' : T_{12}$
by (*rule typing.T-App*)
with *E-App2* **show** *?thesis* **by** *simp*
qed
next
case (*T-TAbs* T_1 Γ t_2 T_2 t')
from ‹$(\lambda<:T_1.\ t_2) \longmapsto t'$›
show *?case* **by** *cases*
next
case (*T-TApp* Γ t_1 T_{11} T_{12} T_2 t')
from ‹$t_1 \cdot_\tau T_2 \longmapsto t'$›
show *?case*
proof *cases*
case (*E-TAbs* T_{11}' t_{12})
with *T-TApp* **have** $\Gamma \vdash (\lambda<:T_{11}'.\ t_{12}) : (\forall<:T_{11}.\ T_{12})$ **by** *simp*
then obtain S'
where *TVarB* $T_{11} :: \Gamma \vdash t_{12} : S'$
and *TVarB* $T_{11} :: \Gamma \vdash S' <: T_{12}$ **by** (*rule TAbs-type'*) *blast*
hence *TVarB* $T_{11} :: \Gamma \vdash t_{12} : T_{12}$ **by** (*rule T-Sub*)
hence $\Gamma \vdash t_{12}[0 \mapsto_\tau T_2] : T_{12}[0 \mapsto_\tau T_2]_\tau$ **using** *T-TApp*(*3*)
by (*rule substT-type* [**where** Δ=[], *simplified*])
with *E-TAbs* **show** *?thesis* **by** *simp*
next
case (*E-TApp* t'')
from ‹$t_1 \longmapsto t''$›
have $\Gamma \vdash t'' : (\forall<:T_{11}.\ T_{12})$ **by** (*rule T-TApp*)
hence $\Gamma \vdash t'' \cdot_\tau T_2 : T_{12}[0 \mapsto_\tau T_2]_\tau$ **using** ‹$\Gamma \vdash T_2 <: T_{11}$›
by (*rule typing.T-TApp*)
with *E-TApp* **show** *?thesis* **by** *simp*
qed
next
case (*T-Sub* Γ t S T t')
from ‹$t \longmapsto t'$›
have $\Gamma \vdash t' : S$ **by** (*rule T-Sub*)
then show *?case* **using** ‹$\Gamma \vdash S <: T$›

by (*rule typing.T-Sub*)
qed

The progress theorem is also proved by induction on the derivation of [] $\vdash$ t : T. In the induction steps, we need the following two lemmas about *canonical forms* stating that closed values of types $T_1 \to T_2$ and $\forall$<:T_1. T_2 must be abstractions over term and type variables, respectively.

lemma *Fun-canonical*: — A.14(1)
assumes *ty*: [] $\vdash$ v : $T_1 \to T_2$
shows $v \in$ *value* $\Longrightarrow \exists\, t\ S.\ v = (\lambda{:}S.\ t)$ **using** *ty*
proof (*induct* []::*env* v $T_1 \to T_2$ *arbitrary*: T_1 T_2)
case *T-Abs*
show *?case* **by** *iprover*
next
case (*T-App* t_1 T_{11} t_2 T_1 T_2)
from ‹$t_1 \cdot t_2 \in$ *value*›
show *?case* **by** *cases*
next
case (*T-TApp* t_1 T_{11} T_{12} T_2 T_1 $T_2{}'$)
from ‹$t_1 \cdot_\tau T_2 \in$ *value*›
show *?case* **by** *cases*
next
case (*T-Sub* t S T_1 T_2)
from ‹[] $\vdash$ S <: $T_1 \to T_2$›
obtain S_1 S_2 **where** S: $S = S_1 \to S_2$
by *cases* (*auto simp add*: *T-Sub*)
show *?case* **by** (*rule T-Sub S*)+
qed *simp*

lemma *TyAll-canonical*: — A.14(3)
assumes *ty*: [] $\vdash$ v : ($\forall$<:T_1. T_2)
shows $v \in$ *value* $\Longrightarrow \exists\, t\ S.\ v = (\lambda{<:}S.\ t)$ **using** *ty*
proof (*induct* []::*env* v $\forall$<:T_1. T_2 *arbitrary*: T_1 T_2)
case (*T-App* t_1 T_{11} t_2 T_1 T_2)
from ‹$t_1 \cdot t_2 \in$ *value*›
show *?case* **by** *cases*
next
case *T-TAbs*
show *?case* **by** *iprover*
next
case (*T-TApp* t_1 T_{11} T_{12} T_2 T_1 $T_2{}'$)
from ‹$t_1 \cdot_\tau T_2 \in$ *value*›
show *?case* **by** *cases*
next
case (*T-Sub* t S T_1 T_2)
from ‹[] $\vdash$ S <: ($\forall$<:T_1. T_2)›
obtain S_1 S_2 **where** S: S = ($\forall$<:S_1. S_2)
by *cases* (*auto simp add*: *T-Sub*)
show *?case* **by** (*rule T-Sub S*)+

```
qed simp

theorem progress:
  assumes ty: [] ⊢ t : T
  shows t ∈ value ∨ (∃ t'. t ⟼ t') using ty
proof (induct []::env t T)
  case T-Var
  thus ?case by simp
next
  case T-Abs
  from value.Abs show ?case ..
next
  case (T-App t1 T11 T12 t2)
  hence t1 ∈ value ∨ (∃ t'. t1 ⟼ t') by simp
  thus ?case
  proof
    assume t1-val: t1 ∈ value
    with T-App obtain t S where t1: t1 = (λ:S. t)
      by (auto dest!: Fun-canonical)
    from T-App have t2 ∈ value ∨ (∃ t'. t2 ⟼ t') by simp
    thus ?thesis
    proof
      assume t2 ∈ value
      with t1 have t1 · t2 ⟼ t[0 ↦ t2]
        by simp (rule eval.intros)
      thus ?thesis by iprover
    next
      assume ∃ t'. t2 ⟼ t'
      then obtain t' where t2 ⟼ t' by iprover
      with t1-val have t1 · t2 ⟼ t1 · t' by (rule eval.intros)
      thus ?thesis by iprover
    qed
  next
    assume ∃ t'. t1 ⟼ t'
    then obtain t' where t1 ⟼ t' ..
    hence t1 · t2 ⟼ t' · t2 by (rule eval.intros)
    thus ?thesis by iprover
  qed
next
  case T-TAbs
  from value.TAbs show ?case ..
next
  case (T-TApp t1 T11 T12 T2)
  hence t1 ∈ value ∨ (∃ t'. t1 ⟼ t') by simp
  thus ?case
  proof
    assume t1 ∈ value
    with T-TApp obtain t S where t1 = (λ<:S. t)
      by (auto dest!: TyAll-canonical)
```

```
    hence t₁ •τ T₂ ⟼ t[0 ↦τ T₂] by simp (rule eval.intros)
    thus ?thesis by iprover
  next
    assume ∃ t′. t₁ ⟼ t′
    then obtain t′ where t₁ ⟼ t′ ..
    hence t₁ •τ T₂ ⟼ t′ •τ T₂ by (rule eval.intros)
    thus ?thesis by iprover
  qed
next
  case (T-Sub t S T)
  show ?case by (rule T-Sub)
qed
```

3 Extending the calculus with records

We now describe how the calculus introduced in the previous section can be extended with records. An important point to note is that many of the definitions and proofs developed for the simple calculus can be reused.

3.1 Types and Terms

In order to represent records, we also need a type of *field names*. For this purpose, we simply use the type of *strings*. We extend the datatype of types of System $F_{<:}$ by a new constructor *RcdT* representing record types.

```
type-synonym name = string

datatype type =
    TVar nat
  | Top
  | Fun type type    (infixr ‹→› 200)
  | TyAll type type  (‹(3∀<:-./ -)› [0, 10] 10)
  | RcdT (name × type) list

type-synonym fldT = name × type
type-synonym rcdT = (name × type) list

datatype binding = VarB type | TVarB type

type-synonym env = binding list

primrec is-TVarB :: binding ⇒ bool
where
  is-TVarB (VarB T) = False
| is-TVarB (TVarB T) = True

primrec type-ofB :: binding ⇒ type
```

where
  type-ofB (*VarB T*) = *T*
| *type-ofB* (*TVarB T*) = *T*

primrec *mapB* :: (*type* ⇒ *type*) ⇒ *binding* ⇒ *binding*
where
  mapB f (*VarB T*) = *VarB* (*f T*)
| *mapB f* (*TVarB T*) = *TVarB* (*f T*)

A record type is essentially an association list, mapping names of record fields to their types. The types of bindings and environments remain unchanged. The datatype *trm* of terms is extended with three new constructors *Rcd*, *Proj*, and *LET*, denoting construction of a new record, selection of a specific field of a record (projection), and matching of a record against a pattern, respectively. A pattern, represented by datatype *pat*, can be either a variable matching any value of a given type, or a nested record pattern. Due to the encoding of variables using de Bruijn indices, a variable pattern only consists of a type.

datatype *pat* = *PVar type* | *PRcd* (*name* × *pat*) *list*

datatype *trm* =
    Var nat
  | *Abs type trm* (‹(3λ:-./ -)› [0, 10] 10)
  | *TAbs type trm* (‹(3λ<:-./ -)› [0, 10] 10)
  | *App trm trm* (**infixl** ‹·› 200)
  | *TApp trm type* (**infixl** ‹·$_\tau$› 200)
  | *Rcd* (*name* × *trm*) *list*
  | *Proj trm name* (‹(-..-)› [90, 91] 90)
  | *LET pat trm trm* (‹(LET (- =/ -)/ IN (-))› 10)

type-synonym *fld* = *name* × *trm*
type-synonym *rcd* = (*name* × *trm*) *list*
type-synonym *fpat* = *name* × *pat*
type-synonym *rpat* = (*name* × *pat*) *list*

In order to motivate the typing and evaluation rules for the *LET*, it is important to note that an expression of the form

$$LET\ PRcd\ [(l_1,\ PVar\ T_1),\ \ldots,\ (l_n,\ PVar\ T_n)] = Rcd\ [(l_1,\ v_1),\ \ldots,\ (l_n,\ v_n)]\ IN\ t$$

can be treated like a nested abstraction $(\lambda{:}T_1.\ \ldots\ \lambda{:}T_n.\ t) \cdot v_1 \cdot \ldots \cdot v_n$

3.2 Lifting and Substitution

primrec *psize* :: *pat* ⇒ *nat* (‹‖-‖$_p$›)
  and *rsize* :: *rpat* ⇒ *nat* (‹‖-‖$_r$›)
  and *fsize* :: *fpat* ⇒ *nat* (‹‖-‖$_f$›)
where

$\|PVar\ T\|_p = 1$
$|\ \|PRcd\ fs\|_p = \|fs\|_r$
$|\ \|[]\|_r = 0$
$|\ \|f :: fs\|_r = \|f\|_f + \|fs\|_r$
$|\ \|(l,\ p)\|_f = \|p\|_p$

primrec $liftT :: nat \Rightarrow nat \Rightarrow type \Rightarrow type\ (\langle\uparrow_\tau\rangle)$
and $liftrT :: nat \Rightarrow nat \Rightarrow rcdT \Rightarrow rcdT\ (\langle\uparrow_{r\tau}\rangle)$
and $liftfT :: nat \Rightarrow nat \Rightarrow fldT \Rightarrow fldT\ (\langle\uparrow_{f\tau}\rangle)$
where
$\uparrow_\tau\ n\ k\ (TVar\ i) = (if\ i < k\ then\ TVar\ i\ else\ TVar\ (i + n))$
$|\ \uparrow_\tau\ n\ k\ Top = Top$
$|\ \uparrow_\tau\ n\ k\ (T \to U) = \uparrow_\tau\ n\ k\ T \to \uparrow_\tau\ n\ k\ U$
$|\ \uparrow_\tau\ n\ k\ (\forall<:T.\ U) = (\forall<:\uparrow_\tau\ n\ k\ T.\ \uparrow_\tau\ n\ (k + 1)\ U)$
$|\ \uparrow_\tau\ n\ k\ (RcdT\ fs) = RcdT\ (\uparrow_{r\tau}\ n\ k\ fs)$
$|\ \uparrow_{r\tau}\ n\ k\ [] = []$
$|\ \uparrow_{r\tau}\ n\ k\ (f :: fs) = \uparrow_{f\tau}\ n\ k\ f :: \uparrow_{r\tau}\ n\ k\ fs$
$|\ \uparrow_{f\tau}\ n\ k\ (l,\ T) = (l,\ \uparrow_\tau\ n\ k\ T)$

primrec $liftp :: nat \Rightarrow nat \Rightarrow pat \Rightarrow pat\ (\langle\uparrow_p\rangle)$
and $liftrp :: nat \Rightarrow nat \Rightarrow rpat \Rightarrow rpat\ (\langle\uparrow_{rp}\rangle)$
and $liftfp :: nat \Rightarrow nat \Rightarrow fpat \Rightarrow fpat\ (\langle\uparrow_{fp}\rangle)$
where
$\uparrow_p\ n\ k\ (PVar\ T) = PVar\ (\uparrow_\tau\ n\ k\ T)$
$|\ \uparrow_p\ n\ k\ (PRcd\ fs) = PRcd\ (\uparrow_{rp}\ n\ k\ fs)$
$|\ \uparrow_{rp}\ n\ k\ [] = []$
$|\ \uparrow_{rp}\ n\ k\ (f :: fs) = \uparrow_{fp}\ n\ k\ f :: \uparrow_{rp}\ n\ k\ fs$
$|\ \uparrow_{fp}\ n\ k\ (l,\ p) = (l,\ \uparrow_p\ n\ k\ p)$

primrec $lift :: nat \Rightarrow nat \Rightarrow trm \Rightarrow trm\ (\langle\uparrow\rangle)$
and $liftr :: nat \Rightarrow nat \Rightarrow rcd \Rightarrow rcd\ (\langle\uparrow_r\rangle)$
and $liftf :: nat \Rightarrow nat \Rightarrow fld \Rightarrow fld\ (\langle\uparrow_f\rangle)$
where
$\uparrow\ n\ k\ (Var\ i) = (if\ i < k\ then\ Var\ i\ else\ Var\ (i + n))$
$|\ \uparrow\ n\ k\ (\lambda:T.\ t) = (\lambda:\uparrow_\tau\ n\ k\ T.\ \uparrow\ n\ (k + 1)\ t)$
$|\ \uparrow\ n\ k\ (\lambda<:T.\ t) = (\lambda<:\uparrow_\tau\ n\ k\ T.\ \uparrow\ n\ (k + 1)\ t)$
$|\ \uparrow\ n\ k\ (s \cdot t) = \uparrow\ n\ k\ s \cdot \uparrow\ n\ k\ t$
$|\ \uparrow\ n\ k\ (t \cdot_\tau\ T) = \uparrow\ n\ k\ t \cdot_\tau \uparrow_\tau\ n\ k\ T$
$|\ \uparrow\ n\ k\ (Rcd\ fs) = Rcd\ (\uparrow_r\ n\ k\ fs)$
$|\ \uparrow\ n\ k\ (t..a) = (\uparrow\ n\ k\ t)..a$
$|\ \uparrow\ n\ k\ (LET\ p = t\ IN\ u) = (LET\ \uparrow_p\ n\ k\ p = \uparrow\ n\ k\ t\ IN\ \uparrow\ n\ (k + \|p\|_p)\ u)$
$|\ \uparrow_r\ n\ k\ [] = []$
$|\ \uparrow_r\ n\ k\ (f :: fs) = \uparrow_f\ n\ k\ f :: \uparrow_r\ n\ k\ fs$
$|\ \uparrow_f\ n\ k\ (l,\ t) = (l,\ \uparrow\ n\ k\ t)$

primrec $substTT :: type \Rightarrow nat \Rightarrow type \Rightarrow type\ \ (\langle\text{-}[\text{-} \mapsto_\tau \text{-}]_\tau\rangle\ [300,\ 0,\ 0]\ 300)$
and $substrTT :: rcdT \Rightarrow nat \Rightarrow type \Rightarrow rcdT\ \ (\langle\text{-}[\text{-} \mapsto_\tau \text{-}]_{r\tau}\rangle\ [300,\ 0,\ 0]\ 300)$
and $substfTT :: fldT \Rightarrow nat \Rightarrow type \Rightarrow fldT\ \ (\langle\text{-}[\text{-} \mapsto_\tau \text{-}]_{f\tau}\rangle\ [300,\ 0,\ 0]\ 300)$
where

$(TVar\ i)[k \mapsto_\tau S]_\tau =$
$\quad (if\ k < i\ then\ TVar\ (i - 1)\ else\ if\ i = k\ then\ \uparrow_\tau k\ 0\ S\ else\ TVar\ i)$
$|\ Top[k \mapsto_\tau S]_\tau = Top$
$|\ (T \to U)[k \mapsto_\tau S]_\tau = T[k \mapsto_\tau S]_\tau \to U[k \mapsto_\tau S]_\tau$
$|\ (\forall <:T.\ U)[k \mapsto_\tau S]_\tau = (\forall <:T[k \mapsto_\tau S]_\tau.\ U[k+1 \mapsto_\tau S]_\tau)$
$|\ (RcdT\ fs)[k \mapsto_\tau S]_\tau = RcdT\ (fs[k \mapsto_\tau S]_{r\tau})$
$|\ [][k \mapsto_\tau S]_{r\tau} = []$
$|\ (f :: fs)[k \mapsto_\tau S]_{r\tau} = f[k \mapsto_\tau S]_{f\tau} :: fs[k \mapsto_\tau S]_{r\tau}$
$|\ (l,\ T)[k \mapsto_\tau S]_{f\tau} = (l,\ T[k \mapsto_\tau S]_\tau)$

primrec $substpT :: pat \Rightarrow nat \Rightarrow type \Rightarrow pat\ \ (\langle\text{-}[\text{-} \mapsto_\tau \text{-}]_p\rangle\ [300,\ 0,\ 0]\ 300)$
and $substrpT :: rpat \Rightarrow nat \Rightarrow type \Rightarrow rpat\ \ (\langle\text{-}[\text{-} \mapsto_\tau \text{-}]_{rp}\rangle\ [300,\ 0,\ 0]\ 300)$
and $substfpT :: fpat \Rightarrow nat \Rightarrow type \Rightarrow fpat\ \ (\langle\text{-}[\text{-} \mapsto_\tau \text{-}]_{fp}\rangle\ [300,\ 0,\ 0]\ 300)$
where
$(PVar\ T)[k \mapsto_\tau S]_p = PVar\ (T[k \mapsto_\tau S]_\tau)$
$|\ (PRcd\ fs)[k \mapsto_\tau S]_p = PRcd\ (fs[k \mapsto_\tau S]_{rp})$
$|\ [][k \mapsto_\tau S]_{rp} = []$
$|\ (f :: fs)[k \mapsto_\tau S]_{rp} = f[k \mapsto_\tau S]_{fp} :: fs[k \mapsto_\tau S]_{rp}$
$|\ (l,\ p)[k \mapsto_\tau S]_{fp} = (l,\ p[k \mapsto_\tau S]_p)$

primrec $decp :: nat \Rightarrow nat \Rightarrow pat \Rightarrow pat\ \ (\langle\downarrow_p\rangle)$
where
$\downarrow_p 0\ k\ p = p$
$|\ \downarrow_p (Suc\ n)\ k\ p = \downarrow_p n\ k\ (p[k \mapsto_\tau Top]_p)$

In addition to the lifting and substitution functions already needed for the basic calculus, we also have to define lifting and substitution functions for patterns, which we denote by $\uparrow_p n\ k\ p$ and $T[k \mapsto_\tau S]_p$, respectively. The extension of the existing lifting and substitution functions to records is fairly standard.

primrec $subst :: trm \Rightarrow nat \Rightarrow trm \Rightarrow trm\ \ (\langle\text{-}[\text{-} \mapsto \text{-}]\rangle\ [300,\ 0,\ 0]\ 300)$
and $substr :: rcd \Rightarrow nat \Rightarrow trm \Rightarrow rcd\ \ (\langle\text{-}[\text{-} \mapsto \text{-}]_r\rangle\ [300,\ 0,\ 0]\ 300)$
and $substf :: fld \Rightarrow nat \Rightarrow trm \Rightarrow fld\ \ (\langle\text{-}[\text{-} \mapsto \text{-}]_f\rangle\ [300,\ 0,\ 0]\ 300)$
where
$(Var\ i)[k \mapsto s] =$
$\quad (if\ k < i\ then\ Var\ (i - 1)\ else\ if\ i = k\ then\ \uparrow k\ 0\ s\ else\ Var\ i)$
$|\ (t \cdot u)[k \mapsto s] = t[k \mapsto s] \cdot u[k \mapsto s]$
$|\ (t \cdot_\tau T)[k \mapsto s] = t[k \mapsto s] \cdot_\tau T[k \mapsto_\tau Top]_\tau$
$|\ (\lambda{:}T.\ t)[k \mapsto s] = (\lambda{:}T[k \mapsto_\tau Top]_\tau.\ t[k+1 \mapsto s])$
$|\ (\lambda<:T.\ t)[k \mapsto s] = (\lambda<:T[k \mapsto_\tau Top]_\tau.\ t[k+1 \mapsto s])$
$|\ (Rcd\ fs)[k \mapsto s] = Rcd\ (fs[k \mapsto s]_r)$
$|\ (t..a)[k \mapsto s] = (t[k \mapsto s])..a$
$|\ (LET\ p = t\ IN\ u)[k \mapsto s] = (LET \downarrow_p 1\ k\ p = t[k \mapsto s]\ IN\ u[k + \|p\|_p \mapsto s])$
$|\ [][k \mapsto s]_r = []$
$|\ (f :: fs)[k \mapsto s]_r = f[k \mapsto s]_f :: fs[k \mapsto s]_r$
$|\ (l,\ t)[k \mapsto s]_f = (l,\ t[k \mapsto s])$

Note that the substitution function on terms is defined simultaneously with a substitution function $fs[k \mapsto s]_r$ on records (i.e. lists of fields), and a substitution function $f[k \mapsto s]_f$ on fields. To avoid conflicts with locally bound variables, we have to add an offset $\|p\|_p$ to k when performing substitution in the body of the LET binder, where $\|p\|_p$ is the number of variables in the pattern p.

primrec $substT :: trm \Rightarrow nat \Rightarrow type \Rightarrow trm$ (‹-[- $\mapsto_\tau$ -]› [300, 0, 0] 300)
  and $substrT :: rcd \Rightarrow nat \Rightarrow type \Rightarrow rcd$ (‹-[- $\mapsto_\tau$ -]$_r$› [300, 0, 0] 300)
  and $substfT :: fld \Rightarrow nat \Rightarrow type \Rightarrow fld$ (‹-[- $\mapsto_\tau$ -]$_f$› [300, 0, 0] 300)
where
  $(Var\ i)[k \mapsto_\tau S] = (if\ k < i\ then\ Var\ (i - 1)\ else\ Var\ i)$
| $(t \cdot u)[k \mapsto_\tau S] = t[k \mapsto_\tau S] \cdot u[k \mapsto_\tau S]$
| $(t \cdot_\tau T)[k \mapsto_\tau S] = t[k \mapsto_\tau S] \cdot_\tau T[k \mapsto_\tau S]_\tau$
| $(\lambda{:}T.\ t)[k \mapsto_\tau S] = (\lambda{:}T[k \mapsto_\tau S]_\tau.\ t[k+1 \mapsto_\tau S])$
| $(\lambda{<:}T.\ t)[k \mapsto_\tau S] = (\lambda{<:}T[k \mapsto_\tau S]_\tau.\ t[k+1 \mapsto_\tau S])$
| $(Rcd\ fs)[k \mapsto_\tau S] = Rcd\ (fs[k \mapsto_\tau S]_r)$
| $(t..a)[k \mapsto_\tau S] = (t[k \mapsto_\tau S])..a$
| $(LET\ p = t\ IN\ u)[k \mapsto_\tau S] =$
    $(LET\ p[k \mapsto_\tau S]_p = t[k \mapsto_\tau S]\ IN\ u[k + \|p\|_p \mapsto_\tau S])$
| $[][k \mapsto_\tau S]_r = []$
| $(f :: fs)[k \mapsto_\tau S]_r = f[k \mapsto_\tau S]_f :: fs[k \mapsto_\tau S]_r$
| $(l,\ t)[k \mapsto_\tau S]_f = (l,\ t[k \mapsto_\tau S])$

primrec $liftE :: nat \Rightarrow nat \Rightarrow env \Rightarrow env$ (‹$\uparrow_e$›)
where
  $\uparrow_e\ n\ k\ [] = []$
| $\uparrow_e\ n\ k\ (B :: \Gamma) = mapB\ (\uparrow_\tau\ n\ (k + \|\Gamma\|))\ B :: \uparrow_e\ n\ k\ \Gamma$

primrec $substE :: env \Rightarrow nat \Rightarrow type \Rightarrow env$ (‹-[- $\mapsto_\tau$ -]$_e$› [300, 0, 0] 300)
where
  $[][k \mapsto_\tau T]_e = []$
| $(B :: \Gamma)[k \mapsto_\tau T]_e = mapB\ (\lambda U.\ U[k + \|\Gamma\| \mapsto_\tau T]_\tau)\ B :: \Gamma[k \mapsto_\tau T]_e$

For the formalization of the reduction rules for LET, we need a function $t[k \mapsto_s us]$ for simultaneously substituting terms us for variables with consecutive indices:

primrec $substs :: trm \Rightarrow nat \Rightarrow trm\ list \Rightarrow trm$ (‹-[- $\mapsto_s$ -]› [300, 0, 0] 300)
where
  $t[k \mapsto_s []] = t$
| $t[k \mapsto_s u :: us] = t[k + \|us\| \mapsto u][k \mapsto_s us]$

primrec $decT :: nat \Rightarrow nat \Rightarrow type \Rightarrow type$ (‹$\downarrow_\tau$›)
where
  $\downarrow_\tau\ 0\ k\ T = T$
| $\downarrow_\tau\ (Suc\ n)\ k\ T = \downarrow_\tau\ n\ k\ (T[k \mapsto_\tau Top]_\tau)$

primrec $decE :: nat \Rightarrow nat \Rightarrow env \Rightarrow env$ (‹$\downarrow_e$›)
where
  $\downarrow_e\ 0\ k\ \Gamma = \Gamma$
| $\downarrow_e\ (Suc\ n)\ k\ \Gamma = \downarrow_e\ n\ k\ (\Gamma[k \mapsto_\tau Top]_e)$

primrec $decrT :: nat \Rightarrow nat \Rightarrow rcdT \Rightarrow rcdT$ $(\langle\downarrow_{r\tau}\rangle)$
where
$\downarrow_{r\tau}\ 0\ k\ fTs = fTs$
$|\ \downarrow_{r\tau}\ (Suc\ n)\ k\ fTs = \downarrow_{r\tau}\ n\ k\ (fTs[k \mapsto_\tau\ Top]_{r\tau})$

The lemmas about substitution and lifting are very similar to those needed for the simple calculus without records, with the difference that most of them have to be proved simultaneously with a suitable property for records.

lemma *liftE-length* [*simp*]: $\|\uparrow_e\ n\ k\ \Gamma\| = \|\Gamma\|$
by (*induct* Γ) *simp-all*

lemma *substE-length* [*simp*]: $\|\Gamma[k \mapsto_\tau\ U]_e\| = \|\Gamma\|$
by (*induct* Γ) *simp-all*

lemma *liftE-nth* [*simp*]:
$(\uparrow_e\ n\ k\ \Gamma)\langle i\rangle = map\text{-}option\ (mapB\ (\uparrow_\tau\ n\ (k + \|\Gamma\| - i - 1)))\ (\Gamma\langle i\rangle)$
by (*induct* Γ *arbitrary*: *i*) (*auto split*: *nat.splits*)

lemma *substE-nth* [*simp*]:
$(\Gamma[0 \mapsto_\tau\ T]_e)\langle i\rangle = map\text{-}option\ (mapB\ (\lambda U.\ U[\|\Gamma\| - i - 1 \mapsto_\tau\ T]_\tau))\ (\Gamma\langle i\rangle)$
by (*induct* Γ *arbitrary*: *i*) (*auto split*: *nat.splits*)

lemma *liftT-liftT* [*simp*]:
$i \le j \Longrightarrow j \le i + m \Longrightarrow \uparrow_\tau\ n\ j\ (\uparrow_\tau\ m\ i\ T) = \uparrow_\tau\ (m + n)\ i\ T$
$i \le j \Longrightarrow j \le i + m \Longrightarrow \uparrow_{r\tau}\ n\ j\ (\uparrow_{r\tau}\ m\ i\ rT) = \uparrow_{r\tau}\ (m + n)\ i\ rT$
$i \le j \Longrightarrow j \le i + m \Longrightarrow \uparrow_{f\tau}\ n\ j\ (\uparrow_{f\tau}\ m\ i\ fT) = \uparrow_{f\tau}\ (m + n)\ i\ fT$
by (*induct* *T* **and** *rT* **and** *fT* *arbitrary*: *i j m n* **and** *i j m n* **and** *i j m n*
rule: *liftT.induct liftrT.induct liftfT.induct*) *simp-all*

lemma *liftT-liftT'* [*simp*]:
$i + m \le j \Longrightarrow \uparrow_\tau\ n\ j\ (\uparrow_\tau\ m\ i\ T) = \uparrow_\tau\ m\ i\ (\uparrow_\tau\ n\ (j - m)\ T)$
$i + m \le j \Longrightarrow \uparrow_{r\tau}\ n\ j\ (\uparrow_{r\tau}\ m\ i\ rT) = \uparrow_{r\tau}\ m\ i\ (\uparrow_{r\tau}\ n\ (j - m)\ rT)$
$i + m \le j \Longrightarrow \uparrow_{f\tau}\ n\ j\ (\uparrow_{f\tau}\ m\ i\ fT) = \uparrow_{f\tau}\ m\ i\ (\uparrow_{f\tau}\ n\ (j - m)\ fT)$
proof (*induct* *T* **and** *rT* **and** *fT* *arbitrary*: *i j m n* **and** *i j m n* **and** *i j m n*
rule: *liftT.induct liftrT.induct liftfT.induct*)
qed (*auto simp*: *Suc-diff-le*)

lemma *lift-size* [*simp*]:
$size\ (\uparrow_\tau\ n\ k\ T) = size\ T$
$size\text{-}list\ (size\text{-}prod\ (\lambda x.\ 0)\ size)\ (\uparrow_{r\tau}\ n\ k\ rT) = size\text{-}list\ (size\text{-}prod\ (\lambda x.\ 0)\ size)$
rT
$size\text{-}prod\ (\lambda x.\ 0)\ size\ (\uparrow_{f\tau}\ n\ k\ fT) = size\text{-}prod\ (\lambda x.\ 0)\ size\ fT$
by (*induct* *T* **and** *rT* **and** *fT* *arbitrary*: *k* **and** *k* **and** *k*
rule: *liftT.induct liftrT.induct liftfT.induct*) *simp-all*

lemma *liftT0* [*simp*]:
$\uparrow_\tau\ 0\ i\ T = T$
$\uparrow_{r\tau}\ 0\ i\ rT = rT$

$\uparrow_{f\tau}$ *0 i fT* = *fT*
by (*induct T* **and** *rT* **and** *fT arbitrary*: *i* **and** *i* **and** *i*
rule: *liftT.induct liftrT.induct liftfT.induct*) *simp-all*

lemma *liftp0* [*simp*]:
$\uparrow_p$ *0 i p* = *p*
$\uparrow_{rp}$ *0 i fs* = *fs*
$\uparrow_{fp}$ *0 i f* = *f*
by (*induct p* **and** *fs* **and** *f arbitrary*: *i* **and** *i* **and** *i*
rule: *liftp.induct liftrp.induct liftfp.induct*) *simp-all*

lemma *lift0* [*simp*]:
$\uparrow$ *0 i t* = *t*
$\uparrow_r$ *0 i fs* = *fs*
$\uparrow_f$ *0 i f* = *f*
by (*induct t* **and** *fs* **and** *f arbitrary*: *i* **and** *i* **and** *i*
rule: *lift.induct liftr.induct liftf.induct*) *simp-all*

theorem *substT-liftT* [*simp*]:
$k \leq k' \Longrightarrow k' < k + n \Longrightarrow (\uparrow_\tau\ n\ k\ T)[k' \mapsto_\tau\ U]_\tau = \uparrow_\tau\ (n - 1)\ k\ T$
$k \leq k' \Longrightarrow k' < k + n \Longrightarrow (\uparrow_{r\tau}\ n\ k\ rT)[k' \mapsto_\tau\ U]_{r\tau} = \uparrow_{r\tau}\ (n - 1)\ k\ rT$
$k \leq k' \Longrightarrow k' < k + n \Longrightarrow (\uparrow_{f\tau}\ n\ k\ fT)[k' \mapsto_\tau\ U]_{f\tau} = \uparrow_{f\tau}\ (n - 1)\ k\ fT$
by (*induct T* **and** *rT* **and** *fT arbitrary*: *k k'* **and** *k k'* **and** *k k'*
rule: *liftT.induct liftrT.induct liftfT.induct*) *simp-all*

theorem *liftT-substT* [*simp*]:
$k \leq k' \Longrightarrow \uparrow_\tau\ n\ k\ (T[k' \mapsto_\tau\ U]_\tau) = \uparrow_\tau\ n\ k\ T[k' + n \mapsto_\tau\ U]_\tau$
$k \leq k' \Longrightarrow \uparrow_{r\tau}\ n\ k\ (rT[k' \mapsto_\tau\ U]_{r\tau}) = \uparrow_{r\tau}\ n\ k\ rT[k' + n \mapsto_\tau\ U]_{r\tau}$
$k \leq k' \Longrightarrow \uparrow_{f\tau}\ n\ k\ (fT[k' \mapsto_\tau\ U]_{f\tau}) = \uparrow_{f\tau}\ n\ k\ fT[k' + n \mapsto_\tau\ U]_{f\tau}$
by (*induct T* **and** *rT* **and** *fT arbitrary*: *k k'* **and** *k k'* **and** *k k'*
rule: *liftT.induct liftrT.induct liftfT.induct*) *auto*

theorem *liftT-substT'* [*simp*]:
$k' < k \Longrightarrow$
$\uparrow_\tau\ n\ k\ (T[k' \mapsto_\tau\ U]_\tau) = \uparrow_\tau\ n\ (k + 1)\ T[k' \mapsto_\tau\ \uparrow_\tau\ n\ (k - k')\ U]_\tau$
$k' < k \Longrightarrow$
$\uparrow_{r\tau}\ n\ k\ (rT[k' \mapsto_\tau\ U]_{r\tau}) = \uparrow_{r\tau}\ n\ (k + 1)\ rT[k' \mapsto_\tau\ \uparrow_\tau\ n\ (k - k')\ U]_{r\tau}$
$k' < k \Longrightarrow$
$\uparrow_{f\tau}\ n\ k\ (fT[k' \mapsto_\tau\ U]_{f\tau}) = \uparrow_{f\tau}\ n\ (k + 1)\ fT[k' \mapsto_\tau\ \uparrow_\tau\ n\ (k - k')\ U]_{f\tau}$
proof (*induct T* **and** *rT* **and** *fT arbitrary*: *k k'* **and** *k k'* **and** *k k'*
rule: *liftT.induct liftrT.induct liftfT.induct*)
qed *auto*

lemma *liftT-substT-Top* [*simp*]:
$k \leq k' \Longrightarrow \uparrow_\tau\ n\ k'\ (T[k \mapsto_\tau\ Top]_\tau) = \uparrow_\tau\ n\ (Suc\ k')\ T[k \mapsto_\tau\ Top]_\tau$
$k \leq k' \Longrightarrow \uparrow_{r\tau}\ n\ k'\ (rT[k \mapsto_\tau\ Top]_{r\tau}) = \uparrow_{r\tau}\ n\ (Suc\ k')\ rT[k \mapsto_\tau\ Top]_{r\tau}$
$k \leq k' \Longrightarrow \uparrow_{f\tau}\ n\ k'\ (fT[k \mapsto_\tau\ Top]_{f\tau}) = \uparrow_{f\tau}\ n\ (Suc\ k')\ fT[k \mapsto_\tau\ Top]_{f\tau}$
proof (*induct T* **and** *rT* **and** *fT arbitrary*: *k k'* **and** *k k'* **and** *k k'*
rule: *liftT.induct liftrT.induct liftfT.induct*)

qed *auto*

theorem *liftE-substE* [*simp*]:
$k \le k' \Longrightarrow \uparrow_e\ n\ k\ (\Gamma[k' \mapsto_\tau U]_e) = \uparrow_e\ n\ k\ \Gamma[k' + n \mapsto_\tau U]_e$
proof (*induct* Γ)
case *Nil*
then show *?case*
by *auto*
next
case (*Cons a* Γ)
then show *?case*
by (*cases a*) (*simp-all add*: *ac-simps*)
qed

lemma *liftT-decT* [*simp*]:
$k \le k' \Longrightarrow \uparrow_\tau\ n\ k'\ (\downarrow_\tau\ m\ k\ T) = \downarrow_\tau\ m\ k\ (\uparrow_\tau\ n\ (m + k')\ T)$
by (*induct m arbitrary*: *T*) *simp-all*

lemma *liftT-substT-strange*:
$\uparrow_\tau\ n\ k\ T[n + k \mapsto_\tau U]_\tau = \uparrow_\tau\ n\ (Suc\ k)\ T[k \mapsto_\tau \uparrow_\tau\ n\ 0\ U]_\tau$
$\uparrow_{r\tau}\ n\ k\ rT[n + k \mapsto_\tau U]_{r\tau} = \uparrow_{r\tau}\ n\ (Suc\ k)\ rT[k \mapsto_\tau \uparrow_\tau\ n\ 0\ U]_{r\tau}$
$\uparrow_{f\tau}\ n\ k\ fT[n + k \mapsto_\tau U]_{f\tau} = \uparrow_{f\tau}\ n\ (Suc\ k)\ fT[k \mapsto_\tau \uparrow_\tau\ n\ 0\ U]_{f\tau}$
proof (*induct T* **and** *rT* **and** *fT arbitrary*: *n k* **and** *n k* **and** *n k*
rule: *liftT.induct liftrT.induct liftfT.induct*)
case (*TyAll x1 x2*)
then show *?case*
by (*metis add.commute add-Suc liftT.simps*(*4*) *plus-1-eq-Suc substTT.simps*(*4*))
qed *auto*

lemma *liftp-liftp* [*simp*]:
$k \le k' \Longrightarrow k' \le k + n \Longrightarrow \uparrow_p\ n'\ k'\ (\uparrow_p\ n\ k\ p) = \uparrow_p\ (n + n')\ k\ p$
$k \le k' \Longrightarrow k' \le k + n \Longrightarrow \uparrow_{rp}\ n'\ k'\ (\uparrow_{rp}\ n\ k\ rp) = \uparrow_{rp}\ (n + n')\ k\ rp$
$k \le k' \Longrightarrow k' \le k + n \Longrightarrow \uparrow_{fp}\ n'\ k'\ (\uparrow_{fp}\ n\ k\ fp) = \uparrow_{fp}\ (n + n')\ k\ fp$
by (*induct p* **and** *rp* **and** *fp arbitrary*: *k k'* **and** *k k'* **and** *k k'*
rule: *liftp.induct liftrp.induct liftfp.induct*) *simp-all*

lemma *liftp-psize*[*simp*]:
$\|\uparrow_p\ n\ k\ p\|_p = \|p\|_p$
$\|\uparrow_{rp}\ n\ k\ fs\|_r = \|fs\|_r$
$\|\uparrow_{fp}\ n\ k\ f\|_f = \|f\|_f$
by (*induct p* **and** *fs* **and** *f rule*: *liftp.induct liftrp.induct liftfp.induct*) *simp-all*

lemma *lift-lift* [*simp*]:
$k \le k' \Longrightarrow k' \le k + n \Longrightarrow \uparrow\ n'\ k'\ (\uparrow\ n\ k\ t) = \uparrow\ (n + n')\ k\ t$
$k \le k' \Longrightarrow k' \le k + n \Longrightarrow \uparrow_r\ n'\ k'\ (\uparrow_r\ n\ k\ fs) = \uparrow_r\ (n + n')\ k\ fs$
$k \le k' \Longrightarrow k' \le k + n \Longrightarrow \uparrow_f\ n'\ k'\ (\uparrow_f\ n\ k\ f) = \uparrow_f\ (n + n')\ k\ f$
by (*induct t* **and** *fs* **and** *f arbitrary*: *k k'* **and** *k k'* **and** *k k'*
rule: *lift.induct liftr.induct liftf.induct*) *simp-all*

lemma *liftE-liftE* [*simp*]:
  $k \le k' \Longrightarrow k' \le k + n \Longrightarrow \uparrow_e\ n'\ k'\ (\uparrow_e\ n\ k\ \Gamma) = \uparrow_e\ (n + n')\ k\ \Gamma$
proof (*induct* Γ *arbitrary*: *k k′*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons a* Γ)
  then show *?case*
    by (*cases a*) *auto*
qed

lemma *liftE-liftE′* [*simp*]:
  $i + m \le j \Longrightarrow \uparrow_e\ n\ j\ (\uparrow_e\ m\ i\ \Gamma) = \uparrow_e\ m\ i\ (\uparrow_e\ n\ (j - m)\ \Gamma)$
proof (*induct* Γ *arbitrary*: *i j m n*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons a* Γ)
  then show *?case*
    by (*cases a*) *auto*
qed

lemma *substT-substT*:
  $i \le j \Longrightarrow$
    $T[Suc\ j \mapsto_\tau V]_\tau[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_\tau = T[i \mapsto_\tau U]_\tau[j \mapsto_\tau V]_\tau$
  $i \le j \Longrightarrow$
    $rT[Suc\ j \mapsto_\tau V]_{r\tau}[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_{r\tau} = rT[i \mapsto_\tau U]_{r\tau}[j \mapsto_\tau V]_{r\tau}$
  $i \le j \Longrightarrow$
    $fT[Suc\ j \mapsto_\tau V]_{f\tau}[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_{f\tau} = fT[i \mapsto_\tau U]_{f\tau}[j \mapsto_\tau V]_{f\tau}$
proof (*induct T* **and** *rT* **and** *fT arbitrary*: *i j U V* **and** *i j U V* **and** *i j U V*
    rule: *liftT.induct liftrT.induct liftfT.induct*)
  case (*TyAll x1 x2*)
  then show *?case*
    by (*metis Suc-eq-plus1 diff-Suc-Suc not-less-eq-eq substTT.simps*(*4*))
qed *auto*

lemma *substT-decT* [*simp*]:
  $k \le j \Longrightarrow (\downarrow_\tau\ i\ k\ T)[j \mapsto_\tau U]_\tau = \downarrow_\tau\ i\ k\ (T[i + j \mapsto_\tau U]_\tau)$
  by (*induct i arbitrary*: *T j*) (*simp-all add*: *substT-substT* [*symmetric*])

lemma *substT-decT′* [*simp*]:
  $i \le j \Longrightarrow \downarrow_\tau\ k\ (Suc\ j)\ T[i \mapsto_\tau Top]_\tau = \downarrow_\tau\ k\ j\ (T[i \mapsto_\tau Top]_\tau)$
  by (*induct k arbitrary*: *i j T*) (*simp-all add*: *substT-substT* [*of - - - - Top*, *simplified*])

lemma *substE-substE*:
  $i \le j \Longrightarrow \Gamma[Suc\ j \mapsto_\tau V]_e[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_e = \Gamma[i \mapsto_\tau U]_e[j \mapsto_\tau V]_e$
proof (*induct* Γ)
  case *Nil*

then show *?case* **by** *auto*
next
case (*Cons a* Γ)
then show *?case*
by (*cases a*) (*simp-all add*: *substT-substT* [*symmetric*])
qed

lemma *substT-decE* [*simp*]:
$i \leq j \Longrightarrow \downarrow_e\ k\ (Suc\ j)\ \Gamma[i \mapsto_\tau\ Top]_e = \downarrow_e\ k\ j\ (\Gamma[i \mapsto_\tau\ Top]_e)$
by (*induct k arbitrary*: *i j* Γ) (*simp-all add*: *substE-substE* [*of - - - - Top*, *simplified*])

lemma *liftE-app* [*simp*]: $\uparrow_e\ n\ k\ (\Gamma\ @\ \Delta) = \uparrow_e\ n\ (k + \|\Delta\|)\ \Gamma\ @\ \uparrow_e\ n\ k\ \Delta$
by (*induct* Γ *arbitrary*: *k*) (*simp-all add*: *ac-simps*)

lemma *substE-app* [*simp*]:
$(\Gamma\ @\ \Delta)[k \mapsto_\tau\ T]_e = \Gamma[k + \|\Delta\| \mapsto_\tau\ T]_e\ @\ \Delta[k \mapsto_\tau\ T]_e$
by (*induct* Γ) (*simp-all add*: *ac-simps*)

lemma *substs-app* [*simp*]: $t[k \mapsto_s\ ts\ @\ us] = t[k + \|us\| \mapsto_s\ ts][k \mapsto_s\ us]$
by (*induct ts arbitrary*: *t k*) (*simp-all add*: *ac-simps*)

theorem *decE-Nil* [*simp*]: $\downarrow_e\ n\ k\ [] = []$
by (*induct n*) *simp-all*

theorem *decE-Cons* [*simp*]:
$\downarrow_e\ n\ k\ (B :: \Gamma) = mapB\ (\downarrow_\tau\ n\ (k + \|\Gamma\|))\ B :: \downarrow_e\ n\ k\ \Gamma$
by (*induct n arbitrary*: *B* Γ; *case-tac B*; *force*)

theorem *decE-app* [*simp*]:
$\downarrow_e\ n\ k\ (\Gamma\ @\ \Delta) = \downarrow_e\ n\ (k + \|\Delta\|)\ \Gamma\ @\ \downarrow_e\ n\ k\ \Delta$
by (*induct n arbitrary*: Γ Δ) *simp-all*

theorem *decT-liftT* [*simp*]:
$k \leq k' \Longrightarrow k' + m \leq k + n \Longrightarrow \downarrow_\tau\ m\ k'\ (\uparrow_\tau\ n\ k\ \Gamma) = \uparrow_\tau\ (n - m)\ k\ \Gamma$
by (*induct m arbitrary*: *n*) *auto*

theorem *decE-liftE* [*simp*]:
$k \leq k' \Longrightarrow k' + m \leq k + n \Longrightarrow \downarrow_e\ m\ k'\ (\uparrow_e\ n\ k\ \Gamma) = \uparrow_e\ (n - m)\ k\ \Gamma$
proof (*induct* Γ *arbitrary*: *k k'*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a* Γ)
then show *?case*
by (*cases a*) *auto*
qed

theorem *liftE0* [*simp*]: $\uparrow_e\ 0\ k\ \Gamma = \Gamma$

proof (*induct* Γ)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons a* Γ)
  then show *?case*
    by (*cases a*) *auto*
qed

lemma *decT-decT* [*simp*]: $\downarrow_\tau\ n\ k\ (\downarrow_\tau\ n'\ (k + n)\ T) = \downarrow_\tau\ (n + n')\ k\ T$
  by (*induct n arbitrary*: *k T*) *simp-all*

lemma *decE-decE* [*simp*]: $\downarrow_e\ n\ k\ (\downarrow_e\ n'\ (k + n)\ \Gamma) = \downarrow_e\ (n + n')\ k\ \Gamma$
  by (*induct n arbitrary*: *k* Γ) *simp-all*

lemma *decE-length* [*simp*]: $\|\downarrow_e\ n\ k\ \Gamma\| = \|\Gamma\|$
  by (*induct* Γ) *simp-all*

lemma *liftrT-assoc-None* [*simp*]: $(\uparrow_{r\tau}\ n\ k\ fs\langle l\rangle_? = \bot) = (fs\langle l\rangle_? = \bot)$
  by (*induct fs rule*: *list.induct*) *auto*

lemma *liftrT-assoc-Some*: $fs\langle l\rangle_? = \lfloor T\rfloor \Longrightarrow \uparrow_{r\tau}\ n\ k\ fs\langle l\rangle_? = \lfloor\uparrow_\tau\ n\ k\ T\rfloor$
  by (*induct fs rule*: *list.induct*) *auto*

lemma *liftrp-assoc-None* [*simp*]: $(\uparrow_{rp}\ n\ k\ fps\langle l\rangle_? = \bot) = (fps\langle l\rangle_? = \bot)$
  by (*induct fps rule*: *list.induct*) *auto*

lemma *liftr-assoc-None* [*simp*]: $(\uparrow_r\ n\ k\ fs\langle l\rangle_? = \bot) = (fs\langle l\rangle_? = \bot)$
  by (*induct fs rule*: *list.induct*) *auto*

lemma *unique-liftrT* [*simp*]: $unique\ (\uparrow_{r\tau}\ n\ k\ fs) = unique\ fs$
  by (*induct fs rule*: *list.induct*) *auto*

lemma *substrTT-assoc-None* [*simp*]: $(fs[k \mapsto_\tau\ U]_{r\tau}\langle a\rangle_? = \bot) = (fs\langle a\rangle_? = \bot)$
  by (*induct fs rule*: *list.induct*) *auto*

lemma *substrTT-assoc-Some* [*simp*]:
  $fs\langle a\rangle_? = \lfloor T\rfloor \Longrightarrow fs[k \mapsto_\tau\ U]_{r\tau}\langle a\rangle_? = \lfloor T[k \mapsto_\tau\ U]_\tau\rfloor$
  by (*induct fs rule*: *list.induct*) *auto*

lemma *substrT-assoc-None* [*simp*]: $(fs[k \mapsto_\tau\ P]_r\langle l\rangle_? = \bot) = (fs\langle l\rangle_? = \bot)$
  by (*induct fs rule*: *list.induct*) *auto*

lemma *substrp-assoc-None* [*simp*]: $(fps[k \mapsto_\tau\ U]_{rp}\langle l\rangle_? = \bot) = (fps\langle l\rangle_? = \bot)$
  by (*induct fps rule*: *list.induct*) *auto*

lemma *substr-assoc-None* [*simp*]: $(fs[k \mapsto u]_r\langle l\rangle_? = \bot) = (fs\langle l\rangle_? = \bot)$
  by (*induct fs rule*: *list.induct*) *auto*

lemma *unique-substrT* [*simp*]: *unique* ($fs[k \mapsto_\tau U]_{r\tau}$) = *unique fs*
  by (*induct fs rule*: *list.induct*) *auto*

lemma *liftrT-set*: $(a,\ T) \in set\ fs \Longrightarrow (a,\ \uparrow_\tau\ n\ k\ T) \in set\ (\uparrow_{r\tau}\ n\ k\ fs)$
  by (*induct fs rule*: *list.induct*) *auto*

lemma *liftrT-setD*:
  $(a,\ T) \in set\ (\uparrow_{r\tau}\ n\ k\ fs) \Longrightarrow \exists\, T'.\ (a,\ T') \in set\ fs \wedge T = \uparrow_\tau\ n\ k\ T'$
  by (*induct fs rule*: *list.induct*) *auto*

lemma *substrT-set*: $(a,\ T) \in set\ fs \Longrightarrow (a,\ T[k \mapsto_\tau U]_\tau) \in set\ (fs[k \mapsto_\tau U]_{r\tau})$
  by (*induct fs rule*: *list.induct*) *auto*

lemma *substrT-setD*:
  $(a,\ T) \in set\ (fs[k \mapsto_\tau U]_{r\tau}) \Longrightarrow \exists\, T'.\ (a,\ T') \in set\ fs \wedge T = T'[k \mapsto_\tau U]_\tau$
  by (*induct fs rule*: *list.induct*) *auto*

3.3 Well-formedness

The definition of well-formedness is extended with a rule stating that a record type *RcdT fs* is well-formed, if for all fields $(l,\ T)$ contained in the list *fs*, the type T is well-formed, and all labels l in *fs* are *unique*.

inductive
  well-formed :: *env* ⇒ *type* ⇒ *bool* (‹- $\vdash_{wf}$ -› [*50*, *50*] *50*)
where
  wf-TVar: $\Gamma\langle i\rangle = \lfloor TVarB\ T\rfloor \Longrightarrow \Gamma \vdash_{wf} TVar\ i$
| *wf-Top*: $\Gamma \vdash_{wf} Top$
| *wf-arrow*: $\Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Gamma \vdash_{wf} T \rightarrow U$
| *wf-all*: $\Gamma \vdash_{wf} T \Longrightarrow TVarB\ T :: \Gamma \vdash_{wf} U \Longrightarrow \Gamma \vdash_{wf} (\forall <: T.\ U)$
| *wf-RcdT*: $unique\ fs \Longrightarrow \forall (l,\ T) \in set\ fs.\ \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} RcdT\ fs$

inductive
  well-formedE :: *env* ⇒ *bool* (‹- $\vdash_{wf}$› [*50*] *50*)
  and *well-formedB* :: *env* ⇒ *binding* ⇒ *bool* (‹- $\vdash_{wfB}$ -› [*50*, *50*] *50*)
where
  $\Gamma \vdash_{wfB} B \equiv \Gamma \vdash_{wf} type\text{-}ofB\ B$
| *wf-Nil*: $[] \vdash_{wf}$
| *wf-Cons*: $\Gamma \vdash_{wfB} B \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow B :: \Gamma \vdash_{wf}$

inductive-cases *well-formed-cases*:
  $\Gamma \vdash_{wf} TVar\ i$
  $\Gamma \vdash_{wf} Top$
  $\Gamma \vdash_{wf} T \rightarrow U$
  $\Gamma \vdash_{wf} (\forall <: T.\ U)$
  $\Gamma \vdash_{wf} (RcdT\ fTs)$

inductive-cases *well-formedE-cases*:
  $B :: \Gamma \vdash_{wf}$

lemma *wf-TVarB*: $\Gamma \vdash_{wf}\ T \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow$ *TVarB* $T :: \Gamma \vdash_{wf}$
  by (*rule wf-Cons*) *simp-all*

lemma *wf-VarB*: $\Gamma \vdash_{wf}\ T \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow$ *VarB* $T :: \Gamma \vdash_{wf}$
  by (*rule wf-Cons*) *simp-all*

lemma *map-is-TVarb*:
  map is-TVarB Γ' = *map is-TVarB* $\Gamma \Longrightarrow$
    $\Gamma\langle i\rangle = \lfloor TVarB\ T \rfloor \Longrightarrow \exists\, T.\ \Gamma'\langle i\rangle = \lfloor TVarB\ T \rfloor$
proof (*induct* Γ *arbitrary*: Γ' T i)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons* a Γ)
  then have $\bigwedge z$. $[\![$*is-TVarB z*$]\!] \Longrightarrow \exists\, T.\ z = TVarB\ T$
    by (*metis binding.exhaust is-TVarB.simps*(*1*))
  with *Cons* **show** *?case* **by** (*auto split*: *nat.split-asm*)
qed

lemma *wf-equallength*:
  assumes *H*: $\Gamma \vdash_{wf}\ T$
  shows *map is-TVarB* Γ' = *map is-TVarB* $\Gamma \Longrightarrow \Gamma' \vdash_{wf}\ T$ **using** *H*
proof (*induct arbitrary*: Γ')
  case (*wf-TVar* Γ i T)
  then show *?case*
    using *map-is-TVarb well-formed.wf-TVar* **by** *blast*
next
  case (*wf-RcdT fs* Γ)
  then show *?case*
    by (*simp add*: *split-beta well-formed.wf-RcdT*)
qed (*fastforce intro*: *well-formed.intros*)+

lemma *wfE-replace*:
  Δ @ $B :: \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wfB}\ B' \Longrightarrow$ *is-TVarB* B' = *is-TVarB* $B \Longrightarrow$
    Δ @ $B' :: \Gamma \vdash_{wf}$
proof (*induct* Δ)
  case *Nil*
  then show *?case*
    by (*metis append-Nil well-formedE-cases wf-Cons*)
next
  case (*Cons* a Δ)
  have $a :: \Delta$ @ $B' :: \Gamma \vdash_{wf}$
  proof (*rule wf-Cons*)
    have §: Δ @ $B :: \Gamma \vdash_{wf}$ Δ @ $B :: \Gamma \vdash_{wfB}\ a$
      using *Cons.prems*(*1*) *well-formedE-cases* **by** *auto*
    with *Cons.prems wf-equallength* **show** Δ @ $B' :: \Gamma \vdash_{wfB}\ a$
      by *auto*
    show Δ @ $B' :: \Gamma \vdash_{wf}$
      by (*simp add*: § *Cons*)

qed
with *Cons well-formedE-cases* **show** *?case* **by** *auto*
qed

lemma *wf-weaken*:
assumes *H*: Δ @ $\Gamma \vdash_{wf}$ *T*
shows $\uparrow_e$ (*Suc 0*) *0* Δ @ *B* :: $\Gamma \vdash_{wf} \uparrow_\tau$ (*Suc 0*) $\|\Delta\|$ *T*
using *H*
proof (*induct* Δ @ Γ *T arbitrary*: Δ)
case (*wf-TVar i T*)
show *?case*
proof (*cases i* < $\|\Delta\|$)
case *True*
with *wf-TVar* **show** *?thesis*
by (*force intro*: *well-formed.wf-TVar*)
next
case *False*
then have *Suc i* − $\|\Delta\|$ = *Suc* (*i* − $\|\Delta\|$)
using *Suc-diff-le leI* **by** *blast*
with *wf-TVar* **show** *?thesis*
by (*force intro*: *well-formed.wf-TVar*)
qed
next
case (*wf-RcdT fs*)
then show *?case*
by (*fastforce dest*: *liftrT-setD intro*: *well-formed.wf-RcdT*)
qed (*fastforce intro*: *well-formed.intros*)+

lemma *wf-weaken'*: $\Gamma \vdash_{wf}$ *T* $\Longrightarrow$ Δ @ $\Gamma \vdash_{wf} \uparrow_\tau$ $\|\Delta\|$ *0 T*
proof (*induct* Δ)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a* Δ)
with *wf-weaken* [*of* []] **show** *?case*
by *fastforce*
qed

lemma *wfE-weaken*: Δ @ $\Gamma \vdash_{wf}$ $\Longrightarrow$ $\Gamma \vdash_{wfB}$ *B* $\Longrightarrow$ $\uparrow_e$ (*Suc 0*) *0* Δ @ *B* :: $\Gamma \vdash_{wf}$
proof (*induct* Δ)
case *Nil*
then show *?case*
by (*simp add*: *wf-Cons*)
next
case (*Cons a* Δ)
show *?case*
proof (*cases a*)
case (*VarB x1*)

with *Cons* **have** $VarB\ (\uparrow_\tau\ (Suc\ 0)\ \|\Delta\|\ x1) :: \uparrow_e\ (Suc\ 0)\ 0\ \Delta\ @\ B :: \Gamma \vdash_{wf}$
by (*metis append-Cons type-ofB.simps*(*1*) *well-formedE-cases wf-Cons wf-weaken*)
with *VarB* **show** *?thesis*
by *simp*
next
case (*TVarB x2*)
with *Cons* **have** $TVarB\ (\uparrow_\tau\ (Suc\ 0)\ \|\Delta\|\ x2) :: \uparrow_e\ (Suc\ 0)\ 0\ \Delta\ @\ B :: \Gamma \vdash_{wf}$
by (*metis append-Cons type-ofB.simps*(*2*) *well-formedE-cases wf-Cons wf-weaken*)
with *TVarB* **show** *?thesis*
by *simp*
qed
qed

lemma *wf-liftB*:
assumes H: $\Gamma \vdash_{wf}$
shows $\Gamma\langle i\rangle = \lfloor VarB\ T\rfloor \Longrightarrow \Gamma \vdash_{wf} \uparrow_\tau\ (Suc\ i)\ 0\ T$
using H
proof (*induct arbitrary*: *i*)
case *wf-Nil*
then show *?case*
by *simp*
next
case (*wf-Cons* Γ *B i*)
show *?case*
proof −
have $VarB\ T :: \Gamma \vdash_{wf} \uparrow_\tau\ (Suc\ 0)\ 0\ T$ **if** $\Gamma \vdash_{wf} T$
by (*metis append-self-conv2 liftE.simps*(*1*) *list.size*(*3*) *wf-weaken that*)
moreover have $B :: \Gamma \vdash_{wf} \uparrow_\tau\ (Suc\ (Suc\ k))\ 0\ T$ **if** $\Gamma\langle k\rangle = \lfloor VarB\ T\rfloor$ **for** k
using *that*
by (*metis One-nat-def Suc-eq-plus1 append-self-conv2 less-eq-nat.simps*(*1*)
liftE.simps(*1*) *liftT-liftT*(*1*) *list.size*(*3*) *wf-Cons.hyps*(*3*) *wf-weaken*)
ultimately show *?thesis*
using *wf-Cons* **by** (*auto split*: *nat.split-asm*)
qed
qed

theorem *wf-subst*:
$\Delta\ @\ B :: \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Delta[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf} T[\|\Delta\| \mapsto_\tau U]_\tau$
$\forall (l,\ T) \in set\ (rT{::}rcdT).\ \Delta\ @\ B :: \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow$
$\forall (l,\ T) \in set\ rT.\ \Delta[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf} T[\|\Delta\| \mapsto_\tau U]_\tau$
$\Delta\ @\ B :: \Gamma \vdash_{wf} snd\ (fT{::}fldT) \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow$
$\Delta[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf} snd\ fT[\|\Delta\| \mapsto_\tau U]_\tau$
proof (*induct T* **and** *rT* **and** *fT arbitrary*: Δ **and** Δ **and** Δ
rule: *liftT.induct liftrT.induct liftfT.induct*)
case (*TVar i* Δ)
show *?case*
proof (*cases* $i \le \|\Delta\|$)
case *True*
with *TVar.prems* **have** $\Delta[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf} \uparrow_\tau\ \|\Delta\|\ 0\ U$

```
      by (metis substE-length wf-weaken')
    with TVar True show ?thesis
      by (auto elim!: well-formed-cases simp add: wf-TVar split: nat.split-asm)
  next
    case False
    then have ‖Δ‖ ≤ i − 1
      by simp
    with TVar False show ?thesis
        by (auto elim!: well-formed-cases simp: wf-TVar split: nat-diff-split-asm
nat.split-asm)
  qed
next
  case Top
  then show ?case
    by (simp add: wf-Top)
next
  case (Fun x1 x2)
  then show ?case
    by (metis substTT.simps(3) well-formed-cases(3) wf-arrow)
next
  case (TyAll type1 type2 Δ)
  then have (TVarB type1 :: Δ)[0 ↦τ U]e @ Γ ⊢wf type2[‖TVarB type1 :: Δ‖
↦τ U]τ
    by (metis append-Cons well-formed-cases(4))
  with TyAll show ?case
    using wf-all by (force simp: elim!: well-formed-cases)
next
  case (RcdT x)
  then show ?case
    by (force simp: intro!: wf-RcdT dest: substrT-setD elim: well-formed-cases)
qed (auto simp: split-beta)

theorem wf-dec: Δ @ Γ ⊢wf T ⟹ Γ ⊢wf ↓τ ‖Δ‖ 0 T
proof (induct Δ arbitrary: T)
  case Nil
  then show ?case by auto
next
  case (Cons a Δ)
  with wf-subst(1) [of []] wf-Top show ?case
    by force
qed

theorem wfE-subst: Δ @ B :: Γ ⊢wf ⟹ Γ ⊢wf U ⟹ Δ[0 ↦τ U]e @ Γ ⊢wf
proof (induct Δ)
  case Nil
  then show ?case
    using well-formedE-cases by auto
next
  case (Cons a Δ)
```

show *?case*
proof (*cases a*)
case (*VarB x*)
with *Cons* **have** *VarB* $(x[\|\Delta\| \mapsto_\tau U]_\tau) :: \Delta[0 \mapsto_\tau U]_e$ @ $\Gamma \vdash_{wf}$
by (*metis append-Cons type-ofB.simps*(*1*) *well-formedE-cases wf-VarB wf-subst*(*1*))
then show *?thesis*
using *VarB* **by** *force*
next
case (*TVarB x*)
with *Cons* **have** *TVarB* $(x[\|\Delta\| \mapsto_\tau U]_\tau) :: \Delta[0 \mapsto_\tau U]_e$ @ $\Gamma \vdash_{wf}$
by (*metis append-Cons type-ofB.simps*(*2*) *well-formedE-cases wf-TVarB wf-subst*(*1*))
with *TVarB* **show** *?thesis*
by *simp*
qed
qed

3.4 Subtyping

The definition of the subtyping judgement is extended with a rule *SA-Rcd* stating that a record type *RcdT fs* is a subtype of *RcdT fs′*, if for all fields (l, T) contained in *fs′*, there exists a corresponding field (l, S) such that S is a subtype of T. If the list *fs′* is empty, *SA-Rcd* can appear as a leaf in the derivation tree of the subtyping judgement. Therefore, the introduction rule needs an additional premise $\Gamma \vdash_{wf}$ to make sure that only subtyping judgements with well-formed contexts are derivable. Moreover, since *fs* can contain additional fields not present in *fs′*, we also have to require that the type *RcdT fs* is well-formed. In order to ensure that the type *RcdT fs′* is well-formed, too, we only have to require that labels in *fs′* are unique, since, by induction on the subtyping derivation, all types contained in *fs′* are already well-formed.

inductive
subtyping :: *env* ⇒ *type* ⇒ *type* ⇒ *bool* (‹- /⊢ - <: -› [*50*, *50*, *50*] *50*)
where
SA-Top: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} S \Longrightarrow \Gamma \vdash S <: Top$
| *SA-refl-TVar*: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} TVar\ i \Longrightarrow \Gamma \vdash TVar\ i <: TVar\ i$
| *SA-trans-TVar*: $\Gamma\langle i\rangle = \lfloor TVarB\ U \rfloor \Longrightarrow$
$\Gamma \vdash \uparrow_\tau (Suc\ i)\ 0\ U <: T \Longrightarrow \Gamma \vdash TVar\ i <: T$
| *SA-arrow*: $\Gamma \vdash T_1 <: S_1 \Longrightarrow \Gamma \vdash S_2 <: T_2 \Longrightarrow \Gamma \vdash S_1 \to S_2 <: T_1 \to T_2$
| *SA-all*: $\Gamma \vdash T_1 <: S_1 \Longrightarrow TVarB\ T_1 :: \Gamma \vdash S_2 <: T_2 \Longrightarrow$
$\Gamma \vdash (\forall <: S_1.\ S_2) <: (\forall <: T_1.\ T_2)$
| *SA-Rcd*: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} RcdT\ fs \Longrightarrow unique\ fs' \Longrightarrow$
$\forall (l, T) \in set\ fs'.\ \exists S.\ (l, S) \in set\ fs \wedge \Gamma \vdash S <: T \Longrightarrow \Gamma \vdash RcdT\ fs <: RcdT\ fs'$

lemma *wf-subtype-env*:
assumes *PQ*: $\Gamma \vdash P <: Q$
shows $\Gamma \vdash_{wf}$ **using** *PQ*

by *induct assumption*+

lemma *wf-subtype*:
assumes *PQ*: $\Gamma \vdash P <: Q$
shows $\Gamma \vdash_{wf} P \wedge \Gamma \vdash_{wf} Q$ **using** *PQ*
by *induct* (*auto intro*: *well-formed.intros elim*!: *wf-equallength*)

lemma *wf-subtypeE*:
assumes *H*: $\Gamma \vdash T <: U$
and *H'*: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow P$
shows *P*
using *H H' wf-subtype wf-subtype-env* **by** *force*

lemma *subtype-refl*: — A.1
$\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash T <: T$
$\Gamma \vdash_{wf} \Longrightarrow \forall (l{::}name,\ T){\in}set\ fTs.\ \Gamma \vdash_{wf} T \longrightarrow \Gamma \vdash T <: T$
$\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} snd\ (fT{::}fldT) \Longrightarrow \Gamma \vdash snd\ fT <: snd\ fT$
by (*induct T* **and** *fTs* **and** *fT arbitrary*: Γ **and** Γ **and** Γ
rule: *liftT.induct liftrT.induct liftfT.induct*,
simp-all add: *split-paired-all*, *simp-all*)
(*blast intro*: *subtyping.intros wf-Nil wf-TVarB elim*: *well-formed-cases*)+

lemma *subtype-weaken*:
assumes *H*: $\Delta\ @\ \Gamma \vdash P <: Q$
and *wf*: $\Gamma \vdash_{wfB} B$
shows $\uparrow_e 1\ 0\ \Delta\ @\ B :: \Gamma \vdash \uparrow_\tau 1\ \|\Delta\|\ P <: \uparrow_\tau 1\ \|\Delta\|\ Q$ **using** *H*
proof (*induct* $\Delta\ @\ \Gamma\ P\ Q$ *arbitrary*: Δ)
case *SA-Top*
with *wf* **show** *?case*
by (*auto intro*: *subtyping.SA-Top wfE-weaken wf-weaken*)
next
case *SA-refl-TVar*
with *wf* **show** *?case*
by (*auto intro*!: *subtyping.SA-refl-TVar wfE-weaken dest*: *wf-weaken*)
next
case (*SA-trans-TVar i U T*)
thus *?case*
proof (*cases* $i < \|\Delta\|$)
case *True*
with *SA-trans-TVar*
have $(\uparrow_e 1\ 0\ \Delta\ @\ B :: \Gamma)\langle i\rangle = \lfloor TVarB\ (\uparrow_\tau 1\ (\|\Delta\| - Suc\ i)\ U)\rfloor$
by *simp*
moreover from *True SA-trans-TVar*
have $\uparrow_e 1\ 0\ \Delta\ @\ B :: \Gamma \vdash$
$\uparrow_\tau (Suc\ i)\ 0\ (\uparrow_\tau 1\ (\|\Delta\| - Suc\ i)\ U) <: \uparrow_\tau 1\ \|\Delta\|\ T$
by *simp*
ultimately have $\uparrow_e 1\ 0\ \Delta\ @\ B :: \Gamma \vdash TVar\ i <: \uparrow_\tau 1\ \|\Delta\|\ T$
by (*rule subtyping.SA-trans-TVar*)
with *True* **show** *?thesis* **by** *simp*

next
case *False*
then have $\mathit{Suc}\ i - \|\Delta\| = \mathit{Suc}\ (i - \|\Delta\|)$ **by** *arith*
with *False SA-trans-TVar* **have** $(\uparrow_e\ \mathit{1}\ \mathit{0}\ \Delta\ @\ B :: \Gamma)\langle \mathit{Suc}\ i\rangle = \lfloor \mathit{TVarB}\ U\rfloor$
by *simp*
moreover from *False SA-trans-TVar*
have $\uparrow_e\ \mathit{1}\ \mathit{0}\ \Delta\ @\ B :: \Gamma \vdash \uparrow_\tau\ (\mathit{Suc}\ (\mathit{Suc}\ i))\ \mathit{0}\ U <: \uparrow_\tau\ \mathit{1}\ \|\Delta\|\ T$
by *simp*
ultimately have $\uparrow_e\ \mathit{1}\ \mathit{0}\ \Delta\ @\ B :: \Gamma \vdash \mathit{TVar}\ (\mathit{Suc}\ i) <: \uparrow_\tau\ \mathit{1}\ \|\Delta\|\ T$
by (*rule subtyping.SA-trans-TVar*)
with *False* **show** *?thesis* **by** *simp*
qed
next
case *SA-arrow*
thus *?case* **by** *simp* (*iprover intro*: *subtyping.SA-arrow*)
next
case $(\mathit{SA\text{-}all}\ T_1\ S_1\ S_2\ T_2)$
with *SA-all*(*4*) [*of TVarB* $T_1 :: \Delta$]
show *?case* **by** *simp* (*iprover intro*: *subtyping.SA-all*)
next
case (*SA-Rcd fs fs′*)
with *wf* **have** $\uparrow_e\ (\mathit{Suc}\ \mathit{0})\ \mathit{0}\ \Delta\ @\ B :: \Gamma \vdash_{wf}$ **by** *simp* (*rule wfE-weaken*)
moreover from ‹$\Delta\ @\ \Gamma \vdash_{wf}\ \mathit{RcdT}\ \mathit{fs}$›
have $\uparrow_e\ (\mathit{Suc}\ \mathit{0})\ \mathit{0}\ \Delta\ @\ B :: \Gamma \vdash_{wf}\ \uparrow_\tau\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ (\mathit{RcdT}\ \mathit{fs})$
by (*rule wf-weaken*)
hence $\uparrow_e\ (\mathit{Suc}\ \mathit{0})\ \mathit{0}\ \Delta\ @\ B :: \Gamma \vdash_{wf}\ \mathit{RcdT}\ (\uparrow_{r\tau}\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ \mathit{fs})$ **by** *simp*
moreover from *SA-Rcd* **have** *unique* $(\uparrow_{r\tau}\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ \mathit{fs}')$ **by** *simp*
moreover have $\forall (l,\ T) \in \mathit{set}\ (\uparrow_{r\tau}\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ \mathit{fs}')$.
$\exists S.\ (l,\ S) \in \mathit{set}\ (\uparrow_{r\tau}\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ \mathit{fs}) \wedge \uparrow_e\ (\mathit{Suc}\ \mathit{0})\ \mathit{0}\ \Delta\ @\ B :: \Gamma \vdash S <: T$
proof (*rule ballpI*)
fix *l T*
assume $(l,\ T) \in \mathit{set}\ (\uparrow_{r\tau}\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ \mathit{fs}')$
then obtain T' **where** $(l,\ T') \in \mathit{set}\ \mathit{fs}'$ **and** T: $T = \uparrow_\tau\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ T'$
by (*blast dest*: *liftrT-setD*)
with *SA-Rcd* **obtain** S **where**
lS: $(l,\ S) \in \mathit{set}\ \mathit{fs}$
and *ST*: $\uparrow_e\ (\mathit{Suc}\ \mathit{0})\ \mathit{0}\ \Delta\ @\ B :: \Gamma \vdash \uparrow_\tau\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ S <: \uparrow_\tau\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|$
$(T[\|\Delta\| \mapsto_\tau\ \mathit{Top}]_\tau)$
by *fastforce*
with T **have** $\uparrow_e\ (\mathit{Suc}\ \mathit{0})\ \mathit{0}\ \Delta\ @\ B :: \Gamma \vdash \uparrow_\tau\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ S <: \uparrow_\tau\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|$
T'
by *simp*
moreover from *lS* **have** $(l,\ \uparrow_\tau\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ S) \in \mathit{set}\ (\uparrow_{r\tau}\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ \mathit{fs})$
by (*rule liftrT-set*)
moreover note T
ultimately show $\exists S.\ (l,\ S) \in \mathit{set}\ (\uparrow_{r\tau}\ (\mathit{Suc}\ \mathit{0})\ \|\Delta\|\ \mathit{fs}) \wedge \uparrow_e\ (\mathit{Suc}\ \mathit{0})\ \mathit{0}\ \Delta\ @\ B$
$:: \Gamma \vdash S <: T$
by *auto*
qed

ultimately have $\uparrow_e$ $(Suc\ 0)\ 0\ \Delta\ @\ B :: \Gamma \vdash RcdT\ (\uparrow_{r\tau}\ (Suc\ 0)\ \|\Delta\|\ fs) <:$
$RcdT\ (\uparrow_{r\tau}\ (Suc\ 0)\ \|\Delta\|\ fs')$
by (*rule subtyping.SA-Rcd*)
thus *?case* **by** *simp*
qed

lemma *subtype-weaken'*: — A.2
$\Gamma \vdash P <: Q \Longrightarrow \Delta\ @\ \Gamma \vdash_{wf} \Longrightarrow \Delta\ @\ \Gamma \vdash \uparrow_\tau\ \|\Delta\|\ 0\ P <: \uparrow_\tau\ \|\Delta\|\ 0\ Q$
proof (*induct* Δ)
case *Nil*
then show *?case*
by *auto*
next
case (*Cons a* Δ)
then have $\Delta\ @\ \Gamma \vdash_{wfB}\ a\ \Delta\ @\ \Gamma \vdash_{wf}$
using *well-formedE-cases* **by** *auto*
with *Cons* **show** *?case*
using *subtype-weaken* [**where** $B=a$ **and** $\Gamma=\Delta\ @\ \Gamma$]
by (*metis Suc-eq-plus1 append-Cons append-Nil bot-nat-0.extremum length-Cons liftE.simps*(*1*) *liftT-liftT*(*1*) *list.size*(*3*))
qed

lemma *fieldT-size* [*simp*]:
$(a,\ T) \in set\ fs \Longrightarrow size\ T < Suc\ (size\text{-}list\ (size\text{-}prod\ (\lambda x.\ 0)\ size)\ fs)$
by (*induct fs arbitrary*: *a T rule*: *list.induct*) *fastforce+*

lemma *subtype-trans*: — A.3
$\Gamma \vdash S <: Q \Longrightarrow \Gamma \vdash Q <: T \Longrightarrow \Gamma \vdash S <: T$
$\Delta\ @\ TVarB\ Q :: \Gamma \vdash M <: N \Longrightarrow \Gamma \vdash P <: Q \Longrightarrow$
$\Delta\ @\ TVarB\ P :: \Gamma \vdash M <: N$
using *wf-measure-size*
proof (*induct Q arbitrary*: $\Gamma\ S\ T\ \Delta\ P\ M\ N$ *rule*: *wf-induct-rule*)
case (*less Q*)
{
fix $\Gamma\ S\ T\ Q'$
assume $\Gamma \vdash S <: Q'$
then have $\Gamma \vdash Q' <: T \Longrightarrow size\ Q = size\ Q' \Longrightarrow \Gamma \vdash S <: T$
proof (*induct arbitrary*: *T*)
case *SA-Top*
from *SA-Top*(*3*) **show** *?case*
by *cases* (*auto intro*: *subtyping.SA-Top SA-Top*)
next
case *SA-refl-TVar* **show** *?case* **by** *fact*
next
case *SA-trans-TVar*
thus *?case* **by** (*auto intro*: *subtyping.SA-trans-TVar*)
next
case (*SA-arrow* $\Gamma\ T_1\ S_1\ S_2\ T_2$)
note *SA-arrow'* = *SA-arrow*

from *SA-arrow*(*5*) **show** *?case*
proof *cases*
case *SA-Top*
with *SA-arrow* **show** *?thesis*
by (*auto intro*: *subtyping.SA-Top wf-arrow elim*: *wf-subtypeE*)
next
case (*SA-arrow* $T_1{}'$ $T_2{}'$)
from *SA-arrow SA-arrow′* **have** $\Gamma \vdash S_1 \rightarrow S_2$ <: $T_1{}' \rightarrow T_2{}'$
by (*auto intro*!: *subtyping.SA-arrow intro*: *less*(*1*) [*of* T_1] *less*(*1*) [*of* T_2])
with *SA-arrow* **show** *?thesis* **by** *simp*
qed
next
case (*SA-all* Γ T_1 S_1 S_2 T_2)
note *SA-all′* = *SA-all*
from *SA-all*(*5*) **show** *?case*
proof *cases*
case *SA-Top*
with *SA-all* **show** *?thesis* **by** (*auto intro*!:
subtyping.SA-Top wf-all intro: *wf-equallength elim*: *wf-subtypeE*)
next
case (*SA-all* $T_1{}'$ $T_2{}'$)
from *SA-all SA-all′* **have** $\Gamma \vdash T_1{}'$ <: S_1
by − (*rule less*(*1*), *simp-all*)
moreover from *SA-all SA-all′* **have** *TVarB* $T_1{}'$ :: $\Gamma \vdash S_2$ <: T_2
by − (*rule less*(*2*) [*of* - [], *simplified*], *simp-all*)
with *SA-all SA-all′* **have** *TVarB* $T_1{}'$ :: $\Gamma \vdash S_2$ <: $T_2{}'$
by − (*rule less*(*1*), *simp-all*)
ultimately have $\Gamma \vdash (\forall$<:$S_1.\ S_2)$ <: $(\forall$<:$T_1{}'.\ T_2{}')$
by (*rule subtyping.SA-all*)
with *SA-all* **show** *?thesis* **by** *simp*
qed
next
case (*SA-Rcd* Γ fs_1 fs_2)
note *SA-Rcd′* = *SA-Rcd*
from *SA-Rcd*(*5*) **show** *?case*
proof *cases*
case *SA-Top*
with *SA-Rcd* **show** *?thesis* **by** (*auto intro*!: *subtyping.SA-Top*)
next
case (*SA-Rcd* $fs_2{}'$)
note ‹$\Gamma \vdash_{wf}$›
moreover note ‹$\Gamma \vdash_{wf}$ *RcdT* fs_1›
moreover note ‹*unique* $fs_2{}'$›
moreover have $\forall (l,\ T) \in$ *set* $fs_2{}'$. $\exists S.\ (l,\ S) \in$ *set* $fs_1 \wedge \Gamma \vdash S$ <: T
proof (*rule ballpI*)
fix *l T*
assume *lT*: $(l,\ T) \in$ *set* $fs_2{}'$
with *SA-Rcd* **obtain** *U* **where**
fs2: $(l,\ U) \in$ *set* fs_2 **and** *UT*: $\Gamma \vdash U$ <: T **by** *blast*

with *SA-Rcd SA-Rcd′* **obtain** *S* **where**
fs1: $(l,\ S) \in set\ fs_1$ **and** *SU*: $\Gamma \vdash S <: U$ **by** *blast*
from *SA-Rcd SA-Rcd′ fs2* **have** $(U,\ Q) \in measure\ size$ **by** *simp*
hence $\Gamma \vdash S <: T$ **using** *SU UT* **by** (*rule less*(*1*))
with *fs1* **show** $\exists S.\ (l,\ S) \in set\ fs_1 \wedge \Gamma \vdash S <: T$ **by** *blast*
qed
ultimately have $\Gamma \vdash RcdT\ fs_1 <: RcdT\ fs_2{}'$ **by** (*rule subtyping.SA-Rcd*)
with *SA-Rcd* **show** *?thesis* **by** *simp*
qed
qed
}
note *tr* = *this*
{
case *1*
thus *?case* **using** *refl* **by** (*rule tr*)
next
case *2*
from *2*(*1*) **show** Δ @ *TVarB P* :: $\Gamma \vdash M <: N$
proof (*induct* Δ @ *TVarB Q* :: Γ *M N arbitrary*: Δ)
case *SA-Top*
with *2* **show** *?case* **by** (*auto intro*!: *subtyping.SA-Top*
intro: *wf-equallength wfE-replace elim*!: *wf-subtypeE*)
next
case *SA-refl-TVar*
with *2* **show** *?case* **by** (*auto intro*!: *subtyping.SA-refl-TVar*
intro: *wf-equallength wfE-replace elim*!: *wf-subtypeE*)
next
case (*SA-trans-TVar i U T*)
show *?case*
proof (*cases* $i < \|\Delta\|$)
case *True*
with *SA-trans-TVar* **show** *?thesis*
by (*auto intro*!: *subtyping.SA-trans-TVar*)
next
case *False*
note *False′* = *False*
show *?thesis*
proof (*cases* $i = \|\Delta\|$)
case *True*
from *SA-trans-TVar* **have** (Δ @ [*TVarB P*]) @ $\Gamma \vdash_{wf}$
by (*auto intro*: *wfE-replace elim*!: *wf-subtypeE*)
with ‹$\Gamma \vdash P <: Q$›
have (Δ @ [*TVarB P*]) @ $\Gamma \vdash \uparrow_\tau \|\Delta$ @ [*TVarB P*]$\|$ *0 P* <: $\uparrow_\tau \|\Delta$ @ [*TVarB P*]$\|$ *0 Q*
by (*rule subtype-weaken′*)
with *SA-trans-TVar True False* **have** Δ @ *TVarB P* :: $\Gamma \vdash \uparrow_\tau (Suc\ \|\Delta\|)$ *0 P* <: *T*
by − (*rule tr*, *simp*+)
with *True* **and** *False* **and** *SA-trans-TVar* **show** *?thesis*

```
        by (auto intro!: subtyping.SA-trans-TVar)
      next
        case False
        with False' have i − ‖Δ‖ = Suc (i − ‖Δ‖ − 1) by arith
        with False False' SA-trans-TVar show ?thesis
          by − (rule subtyping.SA-trans-TVar, simp+)
      qed
    qed
  next
    case SA-arrow
    thus ?case by (auto intro!: subtyping.SA-arrow)
  next
    case (SA-all T₁ S₁ S₂ T₂)
    thus ?case by (auto intro: subtyping.SA-all
      SA-all(4) [of TVarB T₁ :: Δ, simplified])
  next
    case (SA-Rcd fs fs')
    from ‹Γ ⊢ P <: Q› have Γ ⊢wf P by (rule wf-subtypeE)
    with SA-Rcd have Δ @ TVarB P :: Γ ⊢wf
      by − (rule wfE-replace, simp+)
    moreover from SA-Rcd have Δ @ TVarB Q :: Γ ⊢wf RcdT fs by simp
    hence Δ @ TVarB P :: Γ ⊢wf RcdT fs by (rule wf-equallength) simp-all
    moreover note ‹unique fs'›
    moreover from SA-Rcd
    have ∀ (l, T)∈set fs'. ∃ S. (l, S)∈set fs ∧ Δ @ TVarB P :: Γ ⊢ S <: T
      by blast
    ultimately show ?case by (rule subtyping.SA-Rcd)
  qed
}
qed

lemma substT-subtype: — A.10
  assumes H: Δ @ TVarB Q :: Γ ⊢ S <: T
  shows Γ ⊢ P <: Q ⟹ Δ[0 ↦τ P]e @ Γ ⊢ S[‖Δ‖ ↦τ P]τ <: T[‖Δ‖ ↦τ P]τ
  using H
proof (induct Δ @ TVarB Q :: Γ S T arbitrary: Δ)
  case (SA-Top S)
  then show ?case
    by (simp add: subtyping.SA-Top wfE-subst wf-subst(1) wf-subtype)
next
  case (SA-refl-TVar i)
  then show ?case
    by (meson subtype-refl(1) wfE-subst wf-subst(1) wf-subtypeE)
next
  case (SA-trans-TVar i U T Δ)
  have Δ[0 ↦τ P]e @ Γ ⊢ ↑τ ‖Δ‖ 0 P <: T[‖Δ‖ ↦τ P]τ
    if i = ‖Δ‖
  proof −
    have ⟦Δ[0 ↦τ P]e @ Γ ⊢ ↑τ ‖Δ‖ 0 U <: T[‖Δ‖ ↦τ P]τ;
```

$\Delta[\mathit{0} \mapsto_\tau P]_e @ \Gamma \vdash \uparrow_\tau \|\Delta[\mathit{0} \mapsto_\tau P]_e\|\ \mathit{0}\ P <: \uparrow_\tau \|\Delta[\mathit{0} \mapsto_\tau P]_e\|\ \mathit{0}\ U]\!]$
$\Longrightarrow \Delta[\mathit{0} \mapsto_\tau P]_e @ \Gamma \vdash \uparrow_\tau \|\Delta\|\ \mathit{0}\ P <: T[\|\Delta\| \mapsto_\tau P]_\tau$
by (*metis substE-length subtype-trans*(*1*))
then show *?thesis*
using *SA-trans-TVar that wf-subtype-env*
by (*fastforce dest*: *subtype-weaken′* [**where** Γ=Γ **and** Δ=$\Delta[\mathit{0} \mapsto_\tau P]_e$])
qed
moreover have $\Delta[\mathit{0} \mapsto_\tau P]_e @ \Gamma \vdash \mathit{TVar}\ (i - \mathit{Suc}\ \mathit{0}) <: T[\|\Delta\| \mapsto_\tau P]_\tau$
if $\|\Delta\| < i$
proof (*intro subtyping.SA-trans-TVar*)
show $(\Delta[\mathit{0} \mapsto_\tau P]_e @ \Gamma)\langle i - \mathit{Suc}\ \mathit{0}\rangle = \lfloor \mathit{TVarB}\ U\rfloor$
using *SA-trans-TVar that*
by (*auto split*: *nat.split-asm nat-diff-split*)
next
show $\Delta[\mathit{0} \mapsto_\tau P]_e @ \Gamma \vdash \uparrow_\tau (\mathit{Suc}\ (i - \mathit{Suc}\ \mathit{0}))\ \mathit{0}\ U <: T[\|\Delta\| \mapsto_\tau P]_\tau$
using *SA-trans-TVar that* **by** *fastforce*
qed
moreover have $\Delta[\mathit{0} \mapsto_\tau P]_e @ \Gamma \vdash \mathit{TVar}\ i <: T[\|\Delta\| \mapsto_\tau P]_\tau$
if $\|\Delta\| > i$
proof (*intro subtyping.SA-trans-TVar*)
show $(\Delta[\mathit{0} \mapsto_\tau P]_e @ \Gamma)\langle i\rangle = \lfloor \mathit{TVarB}\ (U[\|\Delta\| - \mathit{Suc}\ i \mapsto_\tau P]_\tau)\rfloor$
using *that SA-trans-TVar* **by** (*simp split*: *nat.split-asm nat-diff-split*)
next
show $\Delta[\mathit{0} \mapsto_\tau P]_e @ \Gamma \vdash \uparrow_\tau (\mathit{Suc}\ i)\ \mathit{0}\ (U[\|\Delta\| - \mathit{Suc}\ i \mapsto_\tau P]_\tau) <: T[\|\Delta\|$
$\mapsto_\tau P]_\tau$
using *SA-trans-TVar*
by (*metis Suc-leI zero-le le-add-diff-inverse2 liftT-substT*(*1*) *that*)
qed
ultimately show *?case*
by *auto*
next
case (*SA-arrow* $T_1\ S_1\ S_2\ T_2$)
then show *?case*
by (*simp add*: *subtyping.SA-arrow*)
next
case (*SA-all* $T_1\ S_1\ S_2\ T_2$)
then show *?case*
by (*simp add*: *subtyping.SA-all*)
next
case (*SA-Rcd fs fs′*)
have $\Delta[\mathit{0} \mapsto_\tau P]_e @ \Gamma \vdash_{wf}$
using *SA-Rcd wfE-subst* **by** (*meson wf-subtypeE*)
moreover have $\Delta[\mathit{0} \mapsto_\tau P]_e @ \Gamma \vdash_{wf} \mathit{RcdT}\ (\mathit{fs}[\|\Delta\| \mapsto_\tau P]_{r\tau})$
using *SA-Rcd.hyps*(*2*) *SA-Rcd.prems wf-subst*(*1*) *wf-subtype* **by** *fastforce*
moreover have *unique* $(\mathit{fs}'[\|\Delta\| \mapsto_\tau P]_{r\tau})$
using *SA-Rcd.hyps*(*3*) **by** *auto*
moreover have $\forall (l,\ T) \in \mathit{set}\ (\mathit{fs}'[\|\Delta\| \mapsto_\tau P]_{r\tau}).\ \exists S.\ (l,\ S) \in \mathit{set}\ (\mathit{fs}[\|\Delta\| \mapsto_\tau$
$P]_{r\tau}) \wedge \Delta[\mathit{0} \mapsto_\tau P]_e @ \Gamma \vdash S <: T$
using *SA-Rcd* **by** (*smt* (*verit*) *ballpI case-prodD substrT-set substrT-setD*)

ultimately show *?case*
  by (*simp add*: *subtyping.SA-Rcd*)
qed

lemma *subst-subtype*:
  assumes *H*: Δ @ *VarB V* :: $\Gamma \vdash T <: U$
  shows $\downarrow_e$ *1 0* Δ @ $\Gamma \vdash \downarrow_\tau$ *1* $\|\Delta\|$ *T* <: $\downarrow_\tau$ *1* $\|\Delta\|$ *U*
  using *H*
proof (*induct* Δ @ *VarB V* :: Γ *T U arbitrary*: Δ)
  case (*SA-Top S*)
  then show *?case*
    by (*simp add*: *subtyping.SA-Top wfE-subst wf-Top wf-subst*(*1*))
next
  case (*SA-refl-TVar i*)
  then show *?case*
    by (*metis One-nat-def decE.simps decT.simps subtype-refl*(*1*) *wfE-subst*
      wf-Top wf-subst(*1*))
next
  case (*SA-trans-TVar i U T*)
  then have $*$: $\|\Delta\| > i$
    $\Longrightarrow \Delta[0 \mapsto_\tau Top]_e$ @ Γ
      $\vdash \uparrow_\tau$ (*Suc i*) *0* ($U[\|\Delta\| - Suc\ i \mapsto_\tau Top]_\tau$) <: $T[\|\Delta\| \mapsto_\tau Top]_\tau$
    by (*metis One-nat-def Suc-leI bot-nat-0.extremum decE.simps*(*1*,*2*) *decT.simps*(*1*,*2*)
      le-add-diff-inverse2 liftT-substT(*1*))
  show *?case*
    using *SA-trans-TVar*
    by (*auto simp add*: $*$ *split*: *nat-diff-split intro*!: *subtyping.SA-trans-TVar*)
next
  case (*SA-arrow* T_1 S_1 S_2 T_2)
  then show *?case*
    by (*simp add*: *subtyping.SA-arrow*)
next
  case (*SA-all* T_1 S_1 S_2 T_2)
  then show *?case*
    by (*simp add*: *subtyping.SA-all*)
next
  case (*SA-Rcd fs fs'*)
  have $\Delta[0 \mapsto_\tau Top]_e$ @ $\Gamma \vdash_{wf}$
    using *SA-Rcd.hyps*(*1*) *wfE-subst wf-Top* **by** *auto*
  moreover have $\Delta[0 \mapsto_\tau Top]_e$ @ $\Gamma \vdash_{wf}$ *RcdT* ($fs[\|\Delta\| \mapsto_\tau Top]_{r\tau}$)
    using *SA-Rcd.hyps*(*2*) *wf-Top wf-subst*(*1*) **by** *fastforce*
  moreover have *unique* ($fs'[\|\Delta\| \mapsto_\tau Top]_{r\tau}$)
    by (*simp add*: *SA-Rcd.hyps*)
  moreover have $\forall (l, T) \in set$ ($fs'[\|\Delta\| \mapsto_\tau Top]_{r\tau}$). $\exists S.\ (l, S) \in set$ ($fs[\|\Delta\| \mapsto_\tau$
$Top]_{r\tau}$) $\wedge\ \Delta[0 \mapsto_\tau Top]_e$ @ $\Gamma \vdash S <: T$
    using *SA-Rcd*
    by (*smt* (*verit*) *One-nat-def ballpI case-prodD decE.simps*(*1*,*2*) *decT.simps*(*1*,*2*)
      substrT-set substrT-setD)
  ultimately show *?case*

by (*simp add*: *subtyping.SA-Rcd*)
qed

3.5 Typing

In the formalization of the type checking rule for the *LET* binder, we use an additional judgement $\vdash p : T \Rightarrow \Delta$ for checking whether a given pattern p is compatible with the type T of an object that is to be matched against this pattern. The judgement will be defined simultaneously with a judgement $\vdash ps$ [:] $Ts \Rightarrow \Delta$ for type checking field patterns. Apart from checking the type, the judgement also returns a list of bindings Δ, which can be thought of as a "flattened" list of types of the variables occurring in the pattern. Since typing environments are extended "to the left", the bindings in Δ appear in reverse order.

inductive
  ptyping :: *pat* ⇒ *type* ⇒ *env* ⇒ *bool* (‹⊢ - : - ⇒ -› [*50*, *50*, *50*] *50*)
  and *ptypings* :: *rpat* ⇒ *rcdT* ⇒ *env* ⇒ *bool* (‹⊢ - [:] - ⇒ -› [*50*, *50*, *50*] *50*)
where
  P-Var: $\vdash$ *PVar T* : *T* ⇒ [*VarB T*]
| *P-Rcd*: $\vdash$ *fps* [:] *fTs* ⇒ Δ ⟹ $\vdash$ *PRcd fps* : *RcdT fTs* ⇒ Δ
| *P-Nil*: $\vdash$ [] [:] [] ⇒ []
| *P-Cons*: $\vdash$ *p* : *T* ⇒ Δ_1 ⟹ $\vdash$ *fps* [:] *fTs* ⇒ Δ_2 ⟹ *fps*⟨*l*⟩? = ⊥ ⟹
    $\vdash$ ((*l*, *p*) :: *fps*) [:] ((*l*, *T*) :: *fTs*) ⇒ $\uparrow_e$ ‖Δ_1‖ *0* Δ_2 @ Δ_1

The definition of the typing judgement for terms is extended with the rules *T-Let*, *T-Rcd*, and *T-Proj* for pattern matching, record construction and field selection, respectively. The above typing judgement for patterns is used in the rule *T-Let*. The typing judgement for terms is defined simultaneously with a typing judgement $\Gamma \vdash fs$ [:] *fTs* for record fields.

inductive
  typing :: *env* ⇒ *trm* ⇒ *type* ⇒ *bool* (‹- ⊢ - : -› [*50*, *50*, *50*] *50*)
  and *typings* :: *env* ⇒ *rcd* ⇒ *rcdT* ⇒ *bool* (‹- ⊢ - [:] -› [*50*, *50*, *50*] *50*)
where
  T-Var: $\Gamma \vdash_{wf}$ ⟹ Γ⟨*i*⟩ = ⌊*VarB U*⌋ ⟹ *T* = $\uparrow_\tau$ (*Suc i*) *0* *U* ⟹ Γ ⊢ *Var i* : *T*
| *T-Abs*: *VarB* T_1 :: Γ ⊢ t_2 : T_2 ⟹ Γ ⊢ (λ:T_1. t_2) : T_1 → $\downarrow_\tau$ *1* *0* T_2
| *T-App*: Γ ⊢ t_1 : T_{11} → T_{12} ⟹ Γ ⊢ t_2 : T_{11} ⟹ Γ ⊢ t_1 · t_2 : T_{12}
| *T-TAbs*: *TVarB* T_1 :: Γ ⊢ t_2 : T_2 ⟹ Γ ⊢ (λ<:T_1. t_2) : (∀<:T_1. T_2)
| *T-TApp*: Γ ⊢ t_1 : (∀<:T_{11}. T_{12}) ⟹ Γ ⊢ T_2 <: T_{11} ⟹
    Γ ⊢ t_1 $\cdot_\tau$ T_2 : T_{12}[*0* $\mapsto_\tau$ T_2]$_\tau$
| *T-Sub*: Γ ⊢ *t* : *S* ⟹ Γ ⊢ *S* <: *T* ⟹ Γ ⊢ *t* : *T*
| *T-Let*: Γ ⊢ t_1 : T_1 ⟹ ⊢ *p* : T_1 ⇒ Δ ⟹ Δ @ Γ ⊢ t_2 : T_2 ⟹
    Γ ⊢ (*LET p* = t_1 *IN* t_2) : $\downarrow_\tau$ ‖Δ‖ *0* T_2
| *T-Rcd*: Γ ⊢ *fs* [:] *fTs* ⟹ Γ ⊢ *Rcd fs* : *RcdT fTs*
| *T-Proj*: Γ ⊢ *t* : *RcdT fTs* ⟹ *fTs*⟨*l*⟩? = ⌊*T*⌋ ⟹ Γ ⊢ *t*..*l* : *T*
| *T-Nil*: $\Gamma \vdash_{wf}$ ⟹ Γ ⊢ [] [:] []
| *T-Cons*: Γ ⊢ *t* : *T* ⟹ Γ ⊢ *fs* [:] *fTs* ⟹ *fs*⟨*l*⟩? = ⊥ ⟹
    Γ ⊢ (*l*, *t*) :: *fs* [:] (*l*, *T*) :: *fTs*

theorem *wf-typeE1*:
$\Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wf}$
$\Gamma \vdash fs$ [:] $fTs \Longrightarrow \Gamma \vdash_{wf}$
by (*induct set*: *typing typings*) (*blast elim*: *well-formedE-cases*)+

theorem *wf-typeE2*:
$\Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wf} T$
$\Gamma' \vdash fs$ [:] $fTs \Longrightarrow (\forall (l, T) \in set\ fTs.\ \Gamma' \vdash_{wf} T) \wedge$
unique $fTs \wedge (\forall l.\ (fs\langle l\rangle_? = \bot) = (fTs\langle l\rangle_? = \bot))$
proof (*induct set*: *typing typings*)
case (*T-Abs* T_1 Γ t_2 T_2)
have $\|[]\| = 0$ **and** $\Gamma \vdash_{wf} T_1$
using *T-Abs.hyps*(*1*) *well-formedE-cases wf-typeE1*(*1*) **by** *fastforce*+
then show *?case*
by (*metis One-nat-def T-Abs.hyps*(*2*) *append-Cons append-Nil length-Cons wf-arrow wf-dec*)
next
case (*T-App* Γ t_1 T_{11} T_{12} t_2)
then show *?case*
using *well-formed-cases*(*3*) **by** *blast*
next
case (*T-TAbs* T_1 Γ t_2 T_2)
then show *?case*
by (*metis type-ofB.simps*(*2*) *well-formedE-cases wf-all wf-typeE1*(*1*))
next
case (*T-TApp* Γ t_1 T_{11} T_{12} T_2)
then show *?case*
by (*metis append-Nil length-0-conv substE-length well-formed-cases*(*4*)
wf-subst(*1*) *wf-subtype*)
next
case (*T-Proj* Γ *t fTs l T*)
then show *?case*
by (*metis assoc-set snd-eqD split-beta well-formed-cases*(*5*))
qed (*auto simp*: *wf-subtype wf-dec wf-RcdT wf-liftB*)

lemmas *ptyping-induct* = *ptyping-ptypings.inducts*(*1*)
[*of - - - -* $\lambda x\ y\ z$. *True*, *simplified True-simps*, *consumes 1*,
case-names P-Var P-Rcd]

lemmas *ptypings-induct* = *ptyping-ptypings.inducts*(*2*)
[*of - - -* $\lambda x\ y\ z$. *True*, *simplified True-simps*, *consumes 1*,
case-names P-Nil P-Cons]

lemmas *typing-induct* = *typing-typings.inducts*(*1*)
[*of - - - -* $\lambda x\ y\ z$. *True*, *simplified True-simps*, *consumes 1*,
case-names T-Var T-Abs T-App T-TAbs T-TApp T-Sub T-Let T-Rcd T-Proj]

lemmas *typings-induct* = *typing-typings.inducts*(*2*)

[*of - - -* $\lambda x\ y\ z.$ *True*, *simplified True-simps*, *consumes 1*,
case-names T-Nil T-Cons]

lemma *narrow-type*: — A.7
Δ @ *TVarB Q* :: $\Gamma \vdash t : T \Longrightarrow$
$\Gamma \vdash P <: Q \Longrightarrow \Delta$ @ *TVarB P* :: $\Gamma \vdash t : T$
Δ @ *TVarB Q* :: $\Gamma \vdash$ *ts* [:] *Ts* $\Longrightarrow$
$\Gamma \vdash P <: Q \Longrightarrow \Delta$ @ *TVarB P* :: $\Gamma \vdash$ *ts* [:] *Ts*
proof (*induct* Δ @ *TVarB Q* :: Γ *t T* **and** Δ @ *TVarB Q* :: Γ *ts Ts*
arbitrary: Δ **and** Δ *set*: *typing typings*)
case (*T-Var i U T*)
show *?case*
proof (*intro typing-typings.T-Var*)
show Δ @ *TVarB P* :: $\Gamma \vdash_{wf}$
using *T-Var* **by** (*elim wfE-replace wf-subtypeE*; *simp*)
show (Δ @ *TVarB P* :: Γ)$\langle i \rangle$ = $\lfloor$*VarB U*$\rfloor$
using *T-Var* **by** (*cases* $i < \|\Delta\|$) (*auto split*: *nat.splits*)
next
show $T = \uparrow_\tau$ (*Suc i*) *0 U*
using *T-Var.hyps*(*3*) **by** *blast*
qed
next
case (*T-Abs* T_1 t_2 T_2)
then show *?case*
using *typing-typings.T-Abs* **by** *force*
next
case (*T-TApp* t_1 T_{11} T_{12} T_2)
then show *?case*
using *subtype-trans*(*2*) *typing-typings.T-TApp* **by** *blast*
next
case (*T-Sub t S T*)
then show *?case*
using *subtype-trans*(*2*) *typing-typings.T-Sub* **by** *blast*
next
case *T-Nil*
then show *?case*
by (*metis is-TVarB.simps*(*2*) *type-ofB.simps*(*2*) *typing-typings.T-Nil wfE-replace*
wf-subtypeE)
qed (*auto simp*: *typing-typings.intros*)

lemma *typings-setD*:
assumes *H*: $\Gamma \vdash$ *fs* [:] *fTs*
shows (*l*, *T*) $\in$ *set fTs* $\Longrightarrow \exists t.$ *fs*$\langle l \rangle_?$ = $\lfloor t \rfloor \wedge \Gamma \vdash t : T$
using *H*
by (*induct arbitrary*: *l T rule*: *typings-induct*) *fastforce+*

lemma *subtype-refl'*:
assumes *t*: $\Gamma \vdash t : T$
shows $\Gamma \vdash T <: T$

using *subtype-refl*(*1*) *t wf-typeE1*(*1*) *wf-typeE2*(*1*) **by** *force*

lemma *Abs-type*: — A.13(1)
assumes *H*: $\Gamma \vdash (\lambda{:}S.\ s) : T$
shows $\Gamma \vdash T <: U \to U' \Longrightarrow$
$(\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow \mathit{VarB}\ S :: \Gamma \vdash s : S' \Longrightarrow$
$\Gamma \vdash \downarrow_\tau\ 1\ 0\ S' <: U' \Longrightarrow P) \Longrightarrow P$
using *H*
proof (*induct* Γ $\lambda{:}S.\ s$ *T arbitrary*: $U\ U'\ S\ s\ P$)
case (*T-Abs* $T_1\ \Gamma\ t_2\ T_2$)
from ‹$\Gamma \vdash T_1 \to \downarrow_\tau\ 1\ 0\ T_2 <: U \to U'$›
obtain *ty1*: $\Gamma \vdash U <: T_1$ **and** *ty2*: $\Gamma \vdash \downarrow_\tau\ 1\ 0\ T_2 <: U'$
by *cases simp-all*
from *ty1* ‹$\mathit{VarB}\ T_1 :: \Gamma \vdash t_2 : T_2$› *ty2*
show *?case* **by** (*rule T-Abs*)
next
case (*T-Sub* $\Gamma\ S'\ T$)
from ‹$\Gamma \vdash S' <: T$› **and** ‹$\Gamma \vdash T <: U \to U'$›
have $\Gamma \vdash S' <: U \to U'$ **by** (*rule subtype-trans*(*1*))
then show *?case*
using *T-Sub.hyps*(*2*) *T-Sub.prems*(*2*) **by** *blast*
qed

lemma *Abs-type'*:
assumes $\Gamma \vdash (\lambda{:}S.\ s) : U \to U'$
and $\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow \mathit{VarB}\ S :: \Gamma \vdash s : S' \Longrightarrow \Gamma \vdash \downarrow_\tau\ 1\ 0\ S' <: U' \Longrightarrow P$
shows *P*
using *Abs-type assms subtype-refl'* **by** *blast*

lemma *TAbs-type*: — A.13(2)
assumes *H*: $\Gamma \vdash (\lambda{<:}S.\ s) : T$
shows $\Gamma \vdash T <: (\forall{<:}U.\ U') \Longrightarrow$
$(\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow \mathit{TVarB}\ U :: \Gamma \vdash s : S' \Longrightarrow$
$\mathit{TVarB}\ U :: \Gamma \vdash S' <: U' \Longrightarrow P) \Longrightarrow P$
using *H*
proof (*induct* Γ $\lambda{<:}S.\ s$ *T arbitrary*: $U\ U'\ S\ s\ P$)
case (*T-TAbs* $T_1\ \Gamma\ t_2\ T_2$)
from ‹$\Gamma \vdash (\forall{<:}T_1.\ T_2) <: (\forall{<:}U.\ U')$›
obtain *ty1*: $\Gamma \vdash U <: T_1$ **and** *ty2*: $\mathit{TVarB}\ U :: \Gamma \vdash T_2 <: U'$
by *cases simp-all*
from ‹$\mathit{TVarB}\ T_1 :: \Gamma \vdash t_2 : T_2$›
have $\mathit{TVarB}\ U :: \Gamma \vdash t_2 : T_2$ **using** *ty1*
by (*rule narrow-type* [*of* [], *simplified*])
then show *?case*
using *T-TAbs ty1 ty2* **by** *blast*
next
case (*T-Sub* $\Gamma\ S'\ T$)
from ‹$\Gamma \vdash S' <: T$› **and** ‹$\Gamma \vdash T <: (\forall{<:}U.\ U')$›
have $\Gamma \vdash S' <: (\forall{<:}U.\ U')$ **by** (*rule subtype-trans*(*1*))

with *T-Sub* **show** *?case*
  by *metis*
qed

lemma *TAbs-type'*:
  assumes $\Gamma \vdash (\lambda{<:}S.\ s) : (\forall{<:}U.\ U')$
  and $\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow TVarB\ U :: \Gamma \vdash s : S' \Longrightarrow TVarB\ U :: \Gamma \vdash S' <: U' \Longrightarrow P$
  shows P
  using *assms TAbs-type subtype-refl'* **by** *blast*

In the proof of the preservation theorem, the following elimination rule for typing judgements on record types will be useful:

lemma *Rcd-type1*: — A.13(3)
  assumes $\Gamma \vdash t : T$
  shows $t = Rcd\ fs \Longrightarrow \Gamma \vdash T <: RcdT\ fTs \Longrightarrow$
    $\forall (l,\ U) \in set\ fTs.\ \exists u.\ fs\langle l\rangle_? = \lfloor u\rfloor \wedge \Gamma \vdash u : U$
  using *assms*
proof (*induct arbitrary*: *fs fTs rule*: *typing-induct*)
  case (*T-Sub* Γ *t S T*)
  then show *?case*
    using *subtype-trans*(*1*) **by** *blast*
next
  case (*T-Rcd* Γ *gs gTs*)
  then show *?case*
    by (*force dest*: *typings-setD intro*: *T-Sub elim*: *subtyping.cases*)
qed *blast+*

lemma *Rcd-type1'*:
  assumes *H*: $\Gamma \vdash Rcd\ fs : RcdT\ fTs$
  shows $\forall (l,\ U) \in set\ fTs.\ \exists u.\ fs\langle l\rangle_? = \lfloor u\rfloor \wedge \Gamma \vdash u : U$
  using *H refl subtype-refl'* [*OF H*]
  by (*rule Rcd-type1*)

Intuitively, this means that for a record *Rcd fs* of type *RcdT fTs*, each field with name l associated with a type U in *fTs* must correspond to a field in *fs* with value u, where u has type U. Thanks to the subsumption rule *T-Sub*, the typing judgement for terms is not sensitive to the order of record fields. For example,

$$\Gamma \vdash Rcd\ [(l_1,\ t_1),\ (l_2,\ t_2),\ (l_3,\ t_3)] : RcdT\ [(l_2,\ T_2),\ (l_1,\ T_1)]$$

provided that $\Gamma \vdash t_i : T_i$. Note however that this does not imply

$$\Gamma \vdash [(l_1,\ t_1),\ (l_2,\ t_2),\ (l_3,\ t_3)]\ [:]\ [(l_2,\ T_2),\ (l_1,\ T_1)]$$

In order for this statement to hold, we need to remove the field l_3 and exchange the order of the fields l_1 and l_2. This gives rise to the following variant of the above elimination rule:

```
lemma Rcd-type2-aux:
  ⟦Γ ⊢ T <: RcdT fTs; ∀(l, U)∈set fTs. ∃u. fs⟨l⟩? = ⌊u⌋ ∧ Γ ⊢ u : U⟧
    ⟹ Γ ⊢ map (λ(l, T). (l, the (fs⟨l⟩?))) fTs [:] fTs
proof (induct fTs rule: list.induct)
  case Nil
  then show ?case
    using T-Nil wf-subtypeE by force
next
  case (Cons p list)
  have Γ ⊢ (a, the (fs⟨a⟩?)) :: map (λ(l, T). (l, the (fs⟨l⟩?))) list [:] (a, b) :: list
    if p = (a, b)
    for a b
  proof (rule T-Cons)
    show Γ ⊢ the (fs⟨a⟩?) : b
      using Cons.prems(2) that by auto
    have Γ ⊢ RcdT ((a, b) :: list) <: RcdT list
    proof (intro SA-Rcd)
      show Γ ⊢wf
        using Cons.prems(1) wf-subtypeE by blast
      have *: Γ ⊢wf RcdT (p :: list)
        using Cons.prems(1) wf-subtypeE by blast
      with that show Γ ⊢wf RcdT ((a, b) :: list)
        by auto
      show unique list
        using * well-formed-cases(5) by fastforce
      show ∀(l, T)∈set list. ∃S. (l, S) ∈ set ((a, b) :: list) ∧ Γ ⊢ S <: T
        using Cons.prems(2) subtype-refl' by fastforce
    qed
    with Cons
    show Γ ⊢ map (λ(l, T). (l, the (fs⟨l⟩?))) list [:] list
      by (metis (no-types, lifting) list.set-intros(2) subtype-trans(1) that)
    then show map (λ(l, T). (l, the (fs⟨l⟩?))) list⟨a⟩? = ⊥
      using Cons.prems(1) that well-formed-cases(5) wf-subtype by fastforce
  qed
  then show ?case
    by (auto split: prod.splits)
qed

lemma Rcd-type2:
  Γ ⊢ Rcd fs : T ⟹ Γ ⊢ T <: RcdT fTs ⟹
    Γ ⊢ map (λ(l, T). (l, the (fs⟨l⟩?))) fTs [:] fTs
  by (simp add: Rcd-type1 Rcd-type2-aux)

lemma Rcd-type2':
  assumes H: Γ ⊢ Rcd fs : RcdT fTs
  shows Γ ⊢ map (λ(l, T). (l, the (fs⟨l⟩?))) fTs [:] fTs
  using H subtype-refl' [OF H]
  by (rule Rcd-type2)
```

lemma *T-eq*: $\Gamma \vdash t : T \Longrightarrow T = T' \Longrightarrow \Gamma \vdash t : T'$ **by** *simp*

lemma *ptyping-length* [*simp*]:
$\vdash p : T \Rightarrow \Delta \Longrightarrow \|p\|_p = \|\Delta\|$
$\vdash fps$ [:] $fTs \Rightarrow \Delta \Longrightarrow \|fps\|_r = \|\Delta\|$
by (*induct set*: *ptyping ptypings*) *simp-all*

lemma *lift-ptyping*:
$\vdash p : T \Rightarrow \Delta \Longrightarrow \vdash \uparrow_p n\ k\ p : \uparrow_\tau n\ k\ T \Rightarrow \uparrow_e n\ k\ \Delta$
$\vdash fps$ [:] $fTs \Rightarrow \Delta \Longrightarrow \vdash \uparrow_{rp} n\ k\ fps$ [:] $\uparrow_{r\tau} n\ k\ fTs \Rightarrow \uparrow_e n\ k\ \Delta$
proof (*induct set*: *ptyping ptypings*)
case *P-Nil*
then show *?case*
by (*simp add*: *ptyping-ptypings.P-Nil*)
next
case (*P-Cons p T* Δ_1 *fps fTs* Δ_2 *l*)
then show *?case*
using *P-Cons.hyps*(*2*) *ptyping-ptypings.P-Cons* **by** *fastforce*
qed (*auto simp*: *ptyping.simps*)

lemma *type-weaken*:
$\Delta @ \Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wfB} B \Longrightarrow$
$\uparrow_e 1\ 0\ \Delta @ B :: \Gamma \vdash \uparrow 1\ \|\Delta\|\ t : \uparrow_\tau 1\ \|\Delta\|\ T$
$\Delta @ \Gamma \vdash fs$ [:] $fTs \Longrightarrow \Gamma \vdash_{wfB} B \Longrightarrow$
$\uparrow_e 1\ 0\ \Delta @ B :: \Gamma \vdash \uparrow_r 1\ \|\Delta\|\ fs$ [:] $\uparrow_{r\tau} 1\ \|\Delta\|\ fTs$
proof (*induct* Δ @ Γ *t T* **and** Δ @ Γ *fs fTs arbitrary*: Δ **and** Δ *set*: *typing typings*)
case (*T-Var i U T* Δ)
show *?case*
proof −
have $\uparrow_e (Suc\ 0)\ 0\ \Delta @ B :: \Gamma \vdash Var\ i : \uparrow_\tau (Suc\ 0)\ \|\Delta\|\ T$
if $i < \|\Delta\|$
using *that T-Var* **by** (*force simp*: *typing-typings.T-Var wfE-weaken*)
moreover have $\uparrow_e (Suc\ 0)\ 0\ \Delta @ B :: \Gamma \vdash Var\ (Suc\ i) : \uparrow_\tau (Suc\ 0)\ \|\Delta\|\ T$
if $\neg\ i < \|\Delta\|$
proof (*intro typing-typings.T-Var*)
have ∗: $Suc\ i - \|\Delta\| = Suc\ (i - \|\Delta\|)$
using *that* **by** *simp*
show $\uparrow_e (Suc\ 0)\ 0\ \Delta @ B :: \Gamma \vdash_{wf}$
by (*simp add*: *T-Var wfE-weaken*)
show $(\uparrow_e (Suc\ 0)\ 0\ \Delta @ B :: \Gamma)\langle Suc\ i\rangle = \lfloor VarB\ U \rfloor$
using *T-Var that* **by** (*simp add*: ∗ *split*: *nat.splits*)
show $\uparrow_\tau (Suc\ 0)\ \|\Delta\|\ T = \uparrow_\tau (Suc\ (Suc\ i))\ 0\ U$
using *T-Var.hyps*(*3*) *that* **by** *fastforce*
qed
ultimately show *?thesis*
by *auto*
qed
next

case (*T-Abs* T_1 t_2 T_2)
then show *?case*
using *typing-typings.T-Abs* **by** *force*
next
case (*T-App* t_1 T_{11} T_{12} t_2)
then show *?case*
by (*simp add*: *typing-typings.T-App*)
next
case (*T-TAbs* T_1 t_2 T_2)
then show *?case*
by (*simp add*: *typing-typings.T-TAbs*)
next
case (*T-TApp* t_1 T_{11} T_{12} T_2)
have $\uparrow_e$ (*Suc 0*) *0* Δ @ B :: $\Gamma \vdash \uparrow_\tau$ (*Suc 0*) $\|\Delta\|$ T_2 <: $\uparrow_\tau$ (*Suc 0*) $\|\Delta\|$ T_{11}
using *subtype-weaken* **by** (*simp add*: *T-TApp*)
moreover have $\uparrow_\tau$ (*Suc 0*) (*Suc* $\|\Delta\|$) $T_{12}[0 \mapsto_\tau \uparrow_\tau$ (*Suc 0*) $\|\Delta\|$ $T_2]_\tau = \uparrow_\tau$ (*Suc 0*) $\|\Delta\|$ ($T_{12}[0 \mapsto_\tau T_2]_\tau$)
by (*metis Suc-eq-plus1 add.commute diff-zero le-eq-less-or-eq liftT-substT'*(*1*) *liftT-substT*(*1*) *liftT-substT-strange*(*1*) *not-gr-zero*)
ultimately show *?case*
using *T-TApp*
by (*metis Suc-eq-plus1 add.commute add.right-neutral lift.simps*(*5*) *liftT.simps*(*4*) *typing-typings.T-TApp*)
next
case (*T-Sub t S T*)
then show *?case*
using *subtype-weaken typing-typings.T-Sub* **by** *blast*
next
case (*T-Let* t_1 T_1 p Δ t_2 T_2 Δ')
then have $\uparrow_e$ (*Suc 0*) $\|\Delta'\|$ Δ @ $\uparrow_e$ (*Suc 0*) *0* Δ' @ B :: $\Gamma \vdash \uparrow$ (*Suc 0*) ($\|\Delta\| + \|\Delta'\|$) t_2 : $\uparrow_\tau$ (*Suc 0*) ($\|\Delta\| + \|\Delta'\|$) T_2
by *simp*
with *T-Let*
have $\uparrow_e$ (*Suc 0*) *0* Δ' @ B :: Γ
$\vdash$ (*LET* $\uparrow_p$ (*Suc 0*) $\|\Delta'\|$ p = $\uparrow$ (*Suc 0*) $\|\Delta'\|$ t_1 *IN* $\uparrow$ (*Suc 0*) ($\|\Delta'\| + \|\Delta\|$) t_2) : $\downarrow_\tau$ $\|\Delta\|$ *0* ($\uparrow_\tau$ (*Suc 0*) ($\|\Delta\| + \|\Delta'\|$) T_2)
by (*metis add.commute liftE-length lift-ptyping*(*1*) *nat-1 nat-one-as-int typing-typings.T-Let*)
with *T-Let* **show** *?case*
by (*simp add*: *ac-simps*)
next
case (*T-Rcd fs fTs*)
then show *?case*
by (*simp add*: *typing-typings.T-Rcd*)
next
case (*T-Proj t fTs l T*)
then show *?case*
by (*simp add*: *liftrT-assoc-Some typing-typings.T-Proj*)
next

case *T-Nil*
then show *?case*
by (*simp add*: *typing-typings.T-Nil wfE-weaken*)
next
case (*T-Cons t T fs fTs l*)
then show *?case*
by (*simp add*: *typing-typings.T-Cons*)
qed

lemma *type-weaken'*: — A.5(6)
$\Gamma \vdash t : T \Longrightarrow \Delta \mathbin{@} \Gamma \vdash_{wf} \Longrightarrow \Delta \mathbin{@} \Gamma \vdash \uparrow \|\Delta\|\ 0\ t : \uparrow_\tau \|\Delta\|\ 0\ T$
proof (*induct* Δ)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a* Δ)
then have $\Delta \mathbin{@} \Gamma \vdash_{wfB}\ a\ \Delta \mathbin{@} \Gamma \vdash_{wf}$
by (*auto elim*: *well-formedE-cases*)
with *Cons type-weaken*(*1*)[*of* [], **where** *B*=*a*] **show** *?case*
by (*metis Suc-eq-plus1 append-Cons append-Nil le-add1 le-refl length-Cons liftE.simps*(*1*) *liftT-liftT*(*1*) *lift-lift*(*1*) *list.size*(*3*)))
qed

The substitution lemmas are now proved by mutual induction on the derivations of the typing derivations for terms and lists of fields.

lemma *subst-ptyping*:
$\vdash p : T \Rightarrow \Delta \Longrightarrow \vdash p[k \mapsto_\tau U]_p : T[k \mapsto_\tau U]_\tau \Rightarrow \Delta[k \mapsto_\tau U]_e$
$\vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow \vdash fps[k \mapsto_\tau U]_{rp}\ [:]\ fTs[k \mapsto_\tau U]_{r\tau} \Rightarrow \Delta[k \mapsto_\tau U]_e$
proof (*induct set*: *ptyping ptypings*)
case (*P-Var T*)
then show *?case*
by (*simp add*: *ptyping.simps*)
next
case (*P-Rcd fps fTs* Δ)
then show *?case*
by (*simp add*: *ptyping-ptypings.P-Rcd*)
next
case *P-Nil*
then show *?case*
by (*simp add*: *ptyping-ptypings.P-Nil*)
next
case (*P-Cons p T* Δ_1 *fps fTs* Δ_2 *l*)
then show *?case*
using *ptyping-ptypings.P-Cons* **by** *fastforce*
qed

theorem *subst-type*: — A.8
$\Delta \mathbin{@} VarB\ U :: \Gamma \vdash t : T \Longrightarrow \Gamma \vdash u : U \Longrightarrow$
$\downarrow_e 1\ 0\ \Delta \mathbin{@} \Gamma \vdash t[\|\Delta\| \mapsto u] : \downarrow_\tau 1\ \|\Delta\|\ T$

Δ @ *VarB U* :: $\Gamma \vdash$ *fs* [:] *fTs* $\Longrightarrow \Gamma \vdash u : U \Longrightarrow$
$\downarrow_e$ *1 0* Δ @ $\Gamma \vdash$ *fs*[$\|\Delta\| \mapsto u]_r$ [:] $\downarrow_{r\tau}$ *1* $\|\Delta\|$ *fTs*

proof (*induct* Δ @ *VarB U* :: Γ *t T* **and** Δ @ *VarB U* :: Γ *fs fTs*
arbitrary: Δ **and** Δ *set*: *typing typings*)
case (*T-Var i U' T* Δ')
show *?case*
proof –
have $\Delta'[0 \mapsto_\tau Top]_e$ @ $\Gamma \vdash \uparrow \|\Delta'\|$ *0 u* : $T[\|\Delta'\| \mapsto_\tau Top]_\tau$
if $i = \|\Delta'\|$
using *that T-Var type-weaken' wfE-subst wf-Top* **by** *fastforce*
moreover have $\Delta'[0 \mapsto_\tau Top]_e$ @ $\Gamma \vdash$ *Var* $(i - Suc\ 0)$: $T[\|\Delta'\| \mapsto_\tau Top]_\tau$
if $\|\Delta'\| < i$
proof (*intro typing-typings.T-Var*)
show $\Delta'[0 \mapsto_\tau Top]_e$ @ $\Gamma \vdash_{wf}$
using *T-Var.hyps(1) wfE-subst wf-Top* **by** *force*
have $\|\Delta'\| \leq i - Suc\ 0$
using ‹$\|\Delta'\| < i$› **by** *linarith*
with *T-Var that* **show** $(\Delta'[0 \mapsto_\tau Top]_e$ @ $\Gamma)\langle i - Suc\ 0\rangle = \lfloor VarB\ U'\rfloor$
using *Suc-diff-Suc* **by** (*fastforce simp*: *split*: *nat.split-asm*)
show $T[\|\Delta'\| \mapsto_\tau Top]_\tau = \uparrow_\tau (Suc\ (i - Suc\ 0))\ 0\ U'$
using ‹$\|\Delta'\| < i$› *T-Var.hyps* **by** *auto*
qed
moreover have $\Delta'[0 \mapsto_\tau Top]_e$ @ $\Gamma \vdash$ *Var i* : $T[\|\Delta'\| \mapsto_\tau Top]_\tau$
if $\|\Delta'\| > i$
proof (*intro typing-typings.T-Var*)
show $\Delta'[0 \mapsto_\tau Top]_e$ @ $\Gamma \vdash_{wf}$
using *T-Var wfE-subst wf-Top* **by** *blast*
show $T[\|\Delta'\| \mapsto_\tau Top]_\tau = \uparrow_\tau (Suc\ i)\ 0\ (U'[\|\Delta'\| - Suc\ i \mapsto_\tau Top]_\tau)$
using *T-Var* **by** (*metis that Suc-leI le0 le-add-diff-inverse2 liftT-substT(1)*)
qed (*use that T-Var* **in** *auto*)
ultimately show *?thesis*
by *auto*
qed
next
case (*T-Abs* T_1 t_2 T_2)
then show *?case*
by (*simp add*: *typing-typings.T-Abs* [*THEN T-eq*] *flip*: *substT-substT*)
next
case (*T-TApp* t_1 T_{11} T_{12} T_2)
then show *?case*
using *subst-subtype typing-typings.T-TApp*
apply *simp*
by (*metis diff-zero le0 substT-substT(1) typing-typings.T-TApp*)
next
case (*T-Sub t S T*)
then show *?case*
using *subst-subtype typing-typings.T-Sub* **by** *blast*
next
case (*T-Let* t_1 T_1 *p* Δ t_2 T_2 Δ')

then show *?case*
apply *simp*
by (*metis add.commute substE-length subst-ptyping*(*1*) *typing-typings.T-Let*)
next
case *T-Nil*
then show *?case*
by (*simp add*: *typing-typings.T-Nil wfE-subst wf-Top*)
qed (*auto simp*: *typing-typings.intros*)

theorem *substT-type*: — A.11
Δ @ *TVarB Q* :: $\Gamma \vdash t : T \Longrightarrow \Gamma \vdash P <: Q \Longrightarrow$
$\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash t[\|\Delta\| \mapsto_\tau P] : T[\|\Delta\| \mapsto_\tau P]_\tau$
Δ @ *TVarB Q* :: $\Gamma \vdash fs$ [:] $fTs \Longrightarrow \Gamma \vdash P <: Q \Longrightarrow$
$\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash fs[\|\Delta\| \mapsto_\tau P]_r$ [:] $fTs[\|\Delta\| \mapsto_\tau P]_{r\tau}$
proof (*induct* Δ @ *TVarB Q* :: Γ *t T* **and** Δ @ *TVarB Q* :: Γ *fs fTs*
arbitrary: Δ **and** Δ *set*: *typing typings*)
case (*T-Var i U T* Δ)
show *?case*
proof −
have $\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash_{wf}$
if $\|\Delta\| < i$
using *that*
by (*meson T-Var.hyps*(*1*) *T-Var.prems wfE-subst wf-subtypeE*)
moreover have $(\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma)\langle i - Suc\ 0\rangle = \lfloor VarB\ U \rfloor$
if $\|\Delta\| < i$
using *that T-Var Suc-diff-Suc* **by** (*force split*: *nat.split-asm*)
moreover have $T[\|\Delta\| \mapsto_\tau P]_\tau = \uparrow_\tau (Suc\ (i - Suc\ 0))\ 0\ U$
if $\|\Delta\| < i$
using *that T-Var.hyps* **by** *fastforce*
moreover have $\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash Var\ i : T[\|\Delta\| \mapsto_\tau P]_\tau$
if $\|\Delta\| = i$
using *T-Var that* **by** *auto*
moreover have $\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash Var\ i : T[\|\Delta\| \mapsto_\tau P]_\tau$
if $\|\Delta\| > i$
proof −
have $Suc\ (\|\Delta\| - Suc\ 0) = \|\Delta\|$
using *that* **by** *linarith*
then have §: $\uparrow_\tau (Suc\ i)\ 0\ U[\|\Delta\| \mapsto_\tau P]_\tau = \uparrow_\tau (Suc\ i)\ 0\ (U[\|\Delta\| - Suc\ i$
$\mapsto_\tau P]_\tau)$
using *that* **by** *fastforce*
show *?thesis*
proof (*intro typing-typings.T-Var*)
show $\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash_{wf}$
by (*meson T-Var.hyps*(*1*) *T-Var.prems wfE-subst wf-subtypeE*)
show $(\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma)\langle i\rangle = \lfloor VarB\ (U[\|\Delta\| - Suc\ i \mapsto_\tau P]_\tau)\rfloor$
using § *that T-Var* **by** *simp*
show $T[\|\Delta\| \mapsto_\tau P]_\tau = \uparrow_\tau (Suc\ i)\ 0\ (U[\|\Delta\| - Suc\ i \mapsto_\tau P]_\tau)$
using § *T-Var* **by** *blast*
qed

qed
ultimately show *?thesis*
by (*metis One-nat-def linorder-cases substT.simps*(*1*) *typing-typings.T-Var*)
qed
next
case (*T-Abs* T_1 t_2 T_2)
then show *?case*
by (*simp add*: *typing-typings.T-Abs* [*THEN T-eq*] *flip*: *substT-substT*)
next
case (*T-App* t_1 T_{11} T_{12} t_2)
then show *?case*
using *typing-typings.T-App* **by** *auto*
next
case (*T-TApp* t_1 T_{11} T_{12} T_2)
then show *?case*
apply (*simp add*:)
by (*metis minus-nat.diff-0 substT-substT*(*1*) *substT-subtype typing-typings.T-TApp zero-le*)
next
case (*T-Sub t S T*)
then show *?case*
using *substT-subtype typing-typings.T-Sub* **by** *blast*
next
case (*T-Let* t_1 T_1 p Δ t_2 T_2)
then show *?case*
apply *simp*
by (*metis add.commute substE-length subst-ptyping*(*1*) *typing-typings.T-Let*)
next
case *T-Nil*
then show *?case*
by (*simp add*: *typing-typings.T-Nil wfE-subst wf-subtype*)
qed (*auto simp*: *typing-typings.intros*)

3.6 Evaluation

The definition of canonical values is extended with a clause saying that a record *Rcd fs* is a canonical value if all fields contain canonical values:

inductive-set
value :: *trm set*
where
Abs: $(\lambda{:}T.\ t) \in value$
| *TAbs*: $(\lambda{<:}T.\ t) \in value$
| *Rcd*: $\forall (l,\ t) \in set\ fs.\ t \in value \Longrightarrow Rcd\ fs \in value$

In order to formalize the evaluation rule for *LET*, we introduce another relation $\vdash p \rhd t \Rightarrow ts$ expressing that a pattern p matches a term t. The relation also yields a list of terms ts corresponding to the variables in the pattern. The relation is defined simultaneously with another relation $\vdash fps$

$[\triangleright]$ *fs* $\Rightarrow$ *ts* for matching a list of field patterns *fps* against a list of fields *fs*:

inductive
match :: *pat* ⇒ *trm* ⇒ *trm list* ⇒ *bool* (‹⊢ - ▷ - ⇒ -› [*50*, *50*, *50*] *50*)
and *matchs* :: *rpat* ⇒ *rcd* ⇒ *trm list* ⇒ *bool* (‹⊢ - [▷] - ⇒ -› [*50*, *50*, *50*] *50*)
where
M-PVar: $\vdash PVar\ T \triangleright t \Rightarrow [t]$
| *M-Rcd*: $\vdash fps\ [\triangleright]\ fs \Rightarrow ts \Longrightarrow \vdash PRcd\ fps \triangleright Rcd\ fs \Rightarrow ts$
| *M-Nil*: $\vdash []\ [\triangleright]\ fs \Rightarrow []$
| *M-Cons*: $fs\langle l\rangle_? = \lfloor t\rfloor \Longrightarrow \vdash p \triangleright t \Rightarrow ts \Longrightarrow \vdash fps\ [\triangleright]\ fs \Rightarrow us \Longrightarrow$
$\vdash (l,\ p) :: fps\ [\triangleright]\ fs \Rightarrow ts\ @\ us$

The rules of the evaluation relation for the calculus with records are as follows:

inductive
eval :: *trm* ⇒ *trm* ⇒ *bool* (**infixl** ‹⟼› *50*)
and *evals* :: *rcd* ⇒ *rcd* ⇒ *bool* (**infixl** ‹[⟼]› *50*)
where
E-Abs: $v_2 \in value \Longrightarrow (\lambda{:}T_{11}.\ t_{12}) \cdot v_2 \longmapsto t_{12}[0 \mapsto v_2]$
| *E-TAbs*: $(\lambda{<:}T_{11}.\ t_{12}) \cdot_\tau T_2 \longmapsto t_{12}[0 \mapsto_\tau T_2]$
| *E-App1*: $t \longmapsto t' \Longrightarrow t \cdot u \longmapsto t' \cdot u$
| *E-App2*: $v \in value \Longrightarrow t \longmapsto t' \Longrightarrow v \cdot t \longmapsto v \cdot t'$
| *E-TApp*: $t \longmapsto t' \Longrightarrow t \cdot_\tau T \longmapsto t' \cdot_\tau T$
| *E-LetV*: $v \in value \Longrightarrow \vdash p \triangleright v \Rightarrow ts \Longrightarrow (LET\ p = v\ IN\ t) \longmapsto t[0 \mapsto_s ts]$
| *E-ProjRcd*: $fs\langle l\rangle_? = \lfloor v\rfloor \Longrightarrow v \in value \Longrightarrow Rcd\ fs..l \longmapsto v$
| *E-Proj*: $t \longmapsto t' \Longrightarrow t..l \longmapsto t'..l$
| *E-Rcd*: $fs\ [\longmapsto]\ fs' \Longrightarrow Rcd\ fs \longmapsto Rcd\ fs'$
| *E-Let*: $t \longmapsto t' \Longrightarrow (LET\ p = t\ IN\ u) \longmapsto (LET\ p = t'\ IN\ u)$
| *E-hd*: $t \longmapsto t' \Longrightarrow (l,\ t) :: fs\ [\longmapsto]\ (l,\ t') :: fs$
| *E-tl*: $v \in value \Longrightarrow fs\ [\longmapsto]\ fs' \Longrightarrow (l,\ v) :: fs\ [\longmapsto]\ (l,\ v) :: fs'$

The relation $t \longmapsto t'$ is defined simultaneously with a relation $fs\ [\longmapsto]\ fs'$ for evaluating record fields. The "immediate" reductions, namely pattern matching and projection, are described by the rules *E-LetV* and *E-ProjRcd*, respectively, whereas *E-Proj*, *E-Rcd*, *E-Let*, *E-hd* and *E-tl* are congruence rules.

lemmas *matchs-induct* = *match-matchs.inducts*(*2*)
[*of* - - - *λx y z. True*, *simplified True-simps*, *consumes 1*,
case-names M-Nil M-Cons]

lemmas *evals-induct* = *eval-evals.inducts*(*2*)
[*of* - - *λx y. True*, *simplified True-simps*, *consumes 1*,
case-names E-hd E-tl]

lemma *matchs-mono*:
assumes *H*: $\vdash fps\ [\triangleright]\ fs \Rightarrow ts$
shows $fps\langle l\rangle_? = \bot \Longrightarrow \vdash fps\ [\triangleright]\ (l,\ t) :: fs \Rightarrow ts$
using *H*
proof (*induct rule*: *matchs-induct*)

case (*M-Nil fs*)
then show *?case*
by (*simp add*: *match-matchs.M-Nil*)
next
case (*M-Cons fs l t p ts fps us*)
then show *?case*
by (*metis assoc.simps*(*2*) *fstI match-matchs.M-Cons option.distinct*(*1*))
qed

lemma *matchs-eq*:
assumes *H*: $\vdash$ *fps* [$\rhd$] *fs* $\Rightarrow$ *ts*
shows $\forall\,(l,\ p) \in$ *set fps*. $fs\langle l\rangle_? = fs'\langle l\rangle_? \Longrightarrow \vdash$ *fps* [$\rhd$] $fs' \Rightarrow$ *ts*
using *H*
proof (*induct rule*: *matchs-induct*)
case (*M-Nil fs*)
then show *?case*
using *match-matchs.M-Nil* **by** *auto*
next
case (*M-Cons fs l t p ts fps us*)
then show *?case*
using *match-matchs.M-Cons* **by** *force*
qed

lemma *reorder-eq*:
assumes *H*: $\vdash$ *fps* [:] *fTs* $\Rightarrow \Delta$
shows $\forall\,(l,\ U)\in$*set fTs*. $\exists\,u.\ fs\langle l\rangle_? = \lfloor u\rfloor \Longrightarrow$
$\forall\,(l,\ p) \in$ *set fps*. $fs\langle l\rangle_? = (map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs)\langle l\rangle_?$
using *H* **by** (*induct rule*: *ptypings-induct*) *auto*

lemma *matchs-reorder*:
$\vdash$ *fps* [:] *fTs* $\Rightarrow \Delta \Longrightarrow \forall\,(l,\ U)\in$*set fTs*. $\exists\,u.\ fs\langle l\rangle_? = \lfloor u\rfloor \Longrightarrow$
$\vdash$ *fps* [$\rhd$] *fs* $\Rightarrow$ *ts* $\Longrightarrow \vdash$ *fps* [$\rhd$] $map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs \Rightarrow ts$
by (*rule matchs-eq* [*OF - reorder-eq*], *assumption*+)

lemma *matchs-reorder'*:
$\vdash$ *fps* [:] *fTs* $\Rightarrow \Delta \Longrightarrow \forall\,(l,\ U)\in$*set fTs*. $\exists\,u.\ fs\langle l\rangle_? = \lfloor u\rfloor \Longrightarrow$
$\vdash$ *fps* [$\rhd$] $map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs \Rightarrow ts \Longrightarrow \vdash$ *fps* [$\rhd$] *fs* $\Rightarrow$ *ts*
by (*rule matchs-eq* [*OF - reorder-eq* [*THEN ball-eq-sym*]], *assumption*+)

theorem *matchs-tl*:
assumes *H*: $\vdash$ *fps* [$\rhd$] $(l,\ t)$ :: *fs* $\Rightarrow$ *ts*
shows $fps\langle l\rangle_? = \bot \Longrightarrow \vdash$ *fps* [$\rhd$] *fs* $\Rightarrow$ *ts*
using *H*
proof (*induct fps* $(l,\ t)$ :: *fs ts arbitrary*: *l t fs rule*: *matchs-induct*)
case *M-Nil*
then show *?case*
by (*simp add*: *match-matchs.M-Nil*)
next
case (*M-Cons l t p ts fps us*)

then show *?case*
by (*metis assoc.simps*(*2*) *fst-conv match-matchs.M-Cons not-Some-eq*)
qed

theorem *match-length*:
$\vdash p \rhd t \Rightarrow ts \Longrightarrow \vdash p : T \Rightarrow \Delta \Longrightarrow \|ts\| = \|\Delta\|$
$\vdash fps\ [\rhd]\ ft \Rightarrow ts \Longrightarrow \vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow \|ts\| = \|\Delta\|$
by (*induct arbitrary*: $T\ \Delta$ **and** *fTs* Δ *set*: *match matchs*)
(*erule ptyping.cases ptypings.cases*, *simp*+)+

In the proof of the preservation theorem for the calculus with records, we need the following lemma relating the matching and typing judgements for patterns, which means that well-typed matching preserves typing. Although this property will only be used for $\Gamma_1 = []$ later, the statement must be proved in a more general form in order for the induction to go through.

theorem *match-type*: — A.17
$\vdash p : T_1 \Rightarrow \Delta \Longrightarrow \Gamma_2 \vdash t_1 : T_1 \Longrightarrow$
$\Gamma_1 @ \Delta @ \Gamma_2 \vdash t_2 : T_2 \Longrightarrow \vdash p \rhd t_1 \Rightarrow ts \Longrightarrow$
$\downarrow_e \|\Delta\|\ 0\ \Gamma_1 @ \Gamma_2 \vdash t_2[\|\Gamma_1\| \mapsto_s ts] : \downarrow_\tau \|\Delta\|\ \|\Gamma_1\|\ T_2$
$\vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow \Gamma_2 \vdash fs\ [:]\ fTs \Longrightarrow$
$\Gamma_1 @ \Delta @ \Gamma_2 \vdash t_2 : T_2 \Longrightarrow \vdash fps\ [\rhd]\ fs \Rightarrow ts \Longrightarrow$
$\downarrow_e \|\Delta\|\ 0\ \Gamma_1 @ \Gamma_2 \vdash t_2[\|\Gamma_1\| \mapsto_s ts] : \downarrow_\tau \|\Delta\|\ \|\Gamma_1\|\ T_2$
proof (*induct arbitrary*: $\Gamma_1\ \Gamma_2\ t_1\ t_2\ T_2\ ts$ **and** $\Gamma_1\ \Gamma_2\ fs\ t_2\ T_2\ ts$ *set*: *ptyping ptypings*)
case (*P-Var* $T\ \Gamma_1\ \Gamma_2\ t_1\ t_2\ T_2\ ts$)
from *P-Var* **have** $\Gamma_1[0 \mapsto_\tau Top]_e @ \Gamma_2 \vdash t_2[\|\Gamma_1\| \mapsto t_1] : T_2[\|\Gamma_1\| \mapsto_\tau Top]_\tau$
by − (*rule subst-type* [*simplified*], *simp-all*)
moreover from *P-Var*(*3*) **have** $ts = [t_1]$ **by** *cases simp-all*
ultimately show *?case* **by** *simp*
next
case (*P-Rcd fps fTs* $\Delta\ \Gamma_1\ \Gamma_2\ t_1\ t_2\ T_2\ ts$)
from *P-Rcd*(*5*) **obtain** *fs* **where**
t_1: $t_1 = Rcd\ fs$ **and** *fps*: $\vdash fps\ [\rhd]\ fs \Rightarrow ts$ **by** *cases simp-all*
with *P-Rcd* **have** *fs*: $\Gamma_2 \vdash Rcd\ fs : RcdT\ fTs$ **by** *simp*
hence $\Gamma_2 \vdash map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs\ [:]\ fTs$
by (*rule Rcd-type2'*)
moreover note *P-Rcd*(*4*)
moreover from *fs* **have** $\forall (l,\ U) \in set\ fTs.\ \exists u.\ fs\langle l\rangle_? = \lfloor u\rfloor \wedge \Gamma_2 \vdash u : U$
by (*rule Rcd-type1'*)
hence $\forall (l,\ U) \in set\ fTs.\ \exists u.\ fs\langle l\rangle_? = \lfloor u\rfloor$ **by** *blast*
with *P-Rcd*(*1*) **have** $\vdash fps\ [\rhd]\ map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs \Rightarrow ts$
using *fps* **by** (*rule matchs-reorder*)
ultimately show *?case* **by** (*rule P-Rcd*)
next
case (*P-Nil* $\Gamma_1\ \Gamma_2\ fs\ t_2\ T_2\ ts$)
from *P-Nil*(*3*) **have** $ts = []$ **by** *cases simp-all*
with *P-Nil* **show** *?case* **by** *simp*
next
case (*P-Cons* $p\ T\ \Delta_1\ fps\ fTs\ \Delta_2\ l\ \Gamma_1\ \Gamma_2\ fs\ t_2\ T_2\ ts$)

from *P-Cons*(*8*) **obtain** t ts_1 ts_2 **where**
t: $fs\langle l\rangle_? = \lfloor t\rfloor$ **and** p: $\vdash p \rhd t \Rightarrow ts_1$ **and** *fps*: $\vdash fps\ [\rhd]\ fs \Rightarrow ts_2$
and ts: $ts = ts_1 \ @\ ts_2$ **by** *cases simp-all*
from *P-Cons*(*6*) t *fps* **obtain** fs' **where**
fps': $\vdash fps\ [\rhd]\ (l,\ t) :: fs' \Rightarrow ts_2$ **and** tT: $\Gamma_2 \vdash t : T$ **and** fs': $\Gamma_2 \vdash fs'\ [:]\ fTs$
and l: $fs'\langle l\rangle_? = \bot$ **by** *cases auto*
from *P-Cons* **have** $(\Gamma_1 \ @\ \uparrow_e \|\Delta_1\|\ 0\ \Delta_2) \ @\ \Delta_1 \ @\ \Gamma_2 \vdash t_2 : T_2$ **by** *simp*
with tT **have** ts_1: $\downarrow_e \|\Delta_1\|\ 0\ (\Gamma_1 \ @\ \uparrow_e \|\Delta_1\|\ 0\ \Delta_2) \ @\ \Gamma_2 \vdash$
$t_2[\|\Gamma_1 \ @\ \uparrow_e \|\Delta_1\|\ 0\ \Delta_2\| \mapsto_s ts_1] : \downarrow_\tau \|\Delta_1\|\ \|\Gamma_1 \ @\ \uparrow_e \|\Delta_1\|\ 0\ \Delta_2\|\ T_2$
using p **by** (*rule P-Cons*)
from fps' *P-Cons*(*5*) **have** $\vdash fps\ [\rhd]\ fs' \Rightarrow ts_2$ **by** (*rule matchs-tl*)
with fs' ts_1 [*simplified*]
have $\downarrow_e \|\Delta_2\|\ 0\ (\downarrow_e \|\Delta_1\|\ \|\Delta_2\|\ \Gamma_1) \ @\ \Gamma_2 \vdash t_2[\|\Gamma_1\| + \|\Delta_2\| \mapsto_s ts_1][\|\downarrow_e \|\Delta_1\|$
$\|\Delta_2\|\ \Gamma_1\| \mapsto_s ts_2]$:
$\downarrow_\tau \|\Delta_2\|\ \|\downarrow_e \|\Delta_1\|\ \|\Delta_2\|\ \Gamma_1\|\ (\downarrow_\tau \|\Delta_1\|\ (\|\Gamma_1\| + \|\Delta_2\|)\ T_2)$
by (*rule P-Cons*(*4*))
thus *?case* **by** (*simp add*: *decE-decE* [*of - 0*, *simplified*]
match-length(*2*) [*OF fps P-Cons*(*3*)] *ts*)
qed

lemma *evals-labels* [*simp*]:
assumes H: $fs\ [\longmapsto]\ fs'$
shows $(fs\langle l\rangle_? = \bot) = (fs'\langle l\rangle_? = \bot)$ **using** H
by (*induct rule*: *evals-induct*) *simp-all*

theorem *preservation*: — A.20
$\Gamma \vdash t : T \Longrightarrow t \longmapsto t' \Longrightarrow \Gamma \vdash t' : T$
$\Gamma \vdash fs\ [:]\ fTs \Longrightarrow fs\ [\longmapsto]\ fs' \Longrightarrow \Gamma \vdash fs'\ [:]\ fTs$
proof (*induct arbitrary*: t' **and** fs' *set*: *typing typings*)
case (*T-Var* Γ i U T t')
from ‹*Var* $i \longmapsto t'$›
show *?case* **by** *cases*
next
case (*T-Abs* T_1 Γ t_2 T_2 t')
from ‹$(\lambda{:}T_1.\ t_2) \longmapsto t'$›
show *?case* **by** *cases*
next
case (*T-App* Γ t_1 T_{11} T_{12} t_2 t')
from ‹$t_1 \cdot t_2 \longmapsto t'$›
show *?case*
proof *cases*
case (*E-Abs* T_{11}' t_{12})
with *T-App* **have** $\Gamma \vdash (\lambda{:}T_{11}'.\ t_{12}) : T_{11} \to T_{12}$ **by** *simp*
then obtain S'
where T_{11}: $\Gamma \vdash T_{11} <: T_{11}'$
and t_{12}: *VarB* $T_{11}' :: \Gamma \vdash t_{12} : S'$
and S': $\Gamma \vdash S'[0 \mapsto_\tau Top]_\tau <: T_{12}$ **by** (*rule Abs-type'* [*simplified*]) *blast*
from ‹$\Gamma \vdash t_2 : T_{11}$›
have $\Gamma \vdash t_2 : T_{11}'$ **using** T_{11} **by** (*rule T-Sub*)

with t_{12} **have** $\Gamma \vdash t_{12}[0 \mapsto t_2] : S'[0 \mapsto_\tau Top]_\tau$
  by (*rule subst-type* [**where** Δ=[], *simplified*])
hence $\Gamma \vdash t_{12}[0 \mapsto t_2] : T_{12}$ **using** S' **by** (*rule T-Sub*)
with *E-Abs* **show** *?thesis* **by** *simp*
next
  case (*E-App1* t'')
  from ‹$t_1 \longmapsto t''$›
  have $\Gamma \vdash t'' : T_{11} \to T_{12}$ **by** (*rule T-App*)
  hence $\Gamma \vdash t'' \cdot t_2 : T_{12}$ **using** ‹$\Gamma \vdash t_2 : T_{11}$›
    by (*rule typing-typings.T-App*)
  with *E-App1* **show** *?thesis* **by** *simp*
next
  case (*E-App2* t'')
  from ‹$t_2 \longmapsto t''$›
  have $\Gamma \vdash t'' : T_{11}$ **by** (*rule T-App*)
  with *T-App*(*1*) **have** $\Gamma \vdash t_1 \cdot t'' : T_{12}$
    by (*rule typing-typings.T-App*)
  with *E-App2* **show** *?thesis* **by** *simp*
qed
next
  case (*T-TAbs* T_1 Γ t_2 T_2 t')
  from ‹$(\lambda<:T_1.\ t_2) \longmapsto t'$›
  show *?case* **by** *cases*
next
  case (*T-TApp* Γ t_1 T_{11} T_{12} T_2 t')
  from ‹$t_1 \cdot_\tau T_2 \longmapsto t'$›
  show *?case*
  proof *cases*
    case (*E-TAbs* T_{11}' t_{12})
    with *T-TApp* **have** $\Gamma \vdash (\lambda<:T_{11}'.\ t_{12}) : (\forall<:T_{11}.\ T_{12})$ **by** *simp*
    then obtain S'
      where *TVarB* T_{11} :: $\Gamma \vdash t_{12} : S'$
      and *TVarB* T_{11} :: $\Gamma \vdash S' <: T_{12}$ **by** (*rule TAbs-type'*) *blast*
    hence *TVarB* T_{11} :: $\Gamma \vdash t_{12} : T_{12}$ **by** (*rule T-Sub*)
    hence $\Gamma \vdash t_{12}[0 \mapsto_\tau T_2] : T_{12}[0 \mapsto_\tau T_2]_\tau$ **using** *T-TApp*(*3*)
      by (*rule substT-type* [**where** Δ=[], *simplified*])
    with *E-TAbs* **show** *?thesis* **by** *simp*
  next
    case (*E-TApp* t'')
    from ‹$t_1 \longmapsto t''$›
    have $\Gamma \vdash t'' : (\forall<:T_{11}.\ T_{12})$ **by** (*rule T-TApp*)
    hence $\Gamma \vdash t'' \cdot_\tau T_2 : T_{12}[0 \mapsto_\tau T_2]_\tau$ **using** ‹$\Gamma \vdash T_2 <: T_{11}$›
      by (*rule typing-typings.T-TApp*)
    with *E-TApp* **show** *?thesis* **by** *simp*
  qed
next
  case (*T-Sub* Γ t S T t')
  from ‹$t \longmapsto t'$›
  have $\Gamma \vdash t' : S$ **by** (*rule T-Sub*)

then show *?case* **using** ‹$\Gamma \vdash S <: T$›
  by (*rule typing-typings.T-Sub*)
next
  case (*T-Let* Γ t_1 T_1 p Δ t_2 T_2 t')
  from ‹$(LET\ p = t_1\ IN\ t_2) \longmapsto t'$›
  show *?case*
  proof *cases*
    case (*E-LetV ts*)
    from *T-Let* (*3*,*1*,*4*) ‹$\vdash p \rhd t_1 \Rightarrow ts$›
    have $\Gamma \vdash t_2[0 \mapsto_s ts] : \downarrow_\tau \|\Delta\|\ 0\ T_2$
      by (*rule match-type*(*1*) [*of* - - - - - [], *simplified*])
    with *E-LetV* **show** *?thesis* **by** *simp*
  next
    case (*E-Let* t'')
    from ‹$t_1 \longmapsto t''$›
    have $\Gamma \vdash t'' : T_1$ **by** (*rule T-Let*)
    hence $\Gamma \vdash (LET\ p = t''\ IN\ t_2) : \downarrow_\tau \|\Delta\|\ 0\ T_2$ **using** *T-Let*(*3*,*4*)
      by (*rule typing-typings.T-Let*)
    with *E-Let* **show** *?thesis* **by** *simp*
  qed
next
  case (*T-Rcd* Γ *fs fTs* t')
  from ‹*Rcd fs* $\longmapsto t'$›
  obtain fs' **where** t': $t' = Rcd\ fs'$ **and** *fs*: *fs* [$\longmapsto$] fs'
    by *cases simp-all*
  from *fs* **have** $\Gamma \vdash fs'$ [:] *fTs* **by** (*rule T-Rcd*)
  hence $\Gamma \vdash Rcd\ fs' : RcdT\ fTs$ **by** (*rule typing-typings.T-Rcd*)
  with t' **show** *?case* **by** *simp*
next
  case (*T-Proj* Γ *t fTs l T* t')
  from ‹$t..l \longmapsto t'$›
  show *?case*
  proof *cases*
    case (*E-ProjRcd fs*)
    with *T-Proj* **have** $\Gamma \vdash Rcd\ fs : RcdT\ fTs$ **by** *simp*
    hence $\forall (l, U) \in set\ fTs.\ \exists u.\ fs\langle l\rangle_? = \lfloor u \rfloor \land \Gamma \vdash u : U$
      by (*rule Rcd-type1'*)
    with *E-ProjRcd T-Proj* **show** *?thesis* **by** (*fastforce dest*: *assoc-set*)
  next
    case (*E-Proj* t'')
    from ‹$t \longmapsto t''$›
    have $\Gamma \vdash t'' : RcdT\ fTs$ **by** (*rule T-Proj*)
    hence $\Gamma \vdash t''..l : T$ **using** *T-Proj*(*3*)
      by (*rule typing-typings.T-Proj*)
    with *E-Proj* **show** *?thesis* **by** *simp*
  qed
next
  case (*T-Nil* Γ fs')
  from ‹[] [$\longmapsto$] fs'›

show *?case* **by** *cases*
next
case (*T-Cons* Γ *t* *T* *fs* *fTs* *l* *fs*′)
from ‹$(l,\ t)$ :: *fs* [⟼] *fs*′›
show *?case*
proof *cases*
case (*E-hd* t')
from ‹$t \longmapsto t'$›
have $\Gamma \vdash t' : T$ **by** (*rule T-Cons*)
hence $\Gamma \vdash (l,\ t')$ :: *fs* [:] $(l,\ T)$ :: *fTs* **using** *T-Cons*(*3*,*5*)
by (*rule typing-typings.T-Cons*)
with *E-hd* **show** *?thesis* **by** *simp*
next
case (*E-tl* *fs*″)
note *fs* = ‹*fs* [⟼] *fs*″›
note *T-Cons*(*1*)
moreover from *fs* **have** $\Gamma \vdash$ *fs*″ [:] *fTs* **by** (*rule T-Cons*)
moreover from *fs* *T-Cons* **have** *fs*″$\langle l \rangle_?$ = ⊥ **by** *simp*
ultimately have $\Gamma \vdash (l,\ t)$ :: *fs*″ [:] $(l,\ T)$ :: *fTs*
by (*rule typing-typings.T-Cons*)
with *E-tl* **show** *?thesis* **by** *simp*
qed
qed

lemma *Fun-canonical*: — A.14(1)
assumes *ty*: $[] \vdash v : T_1 \rightarrow T_2$
shows $v \in value \Longrightarrow \exists\, t\ S.\ v = (\lambda{:}S.\ t)$ **using** *ty*
proof (*induct* []::*env* v $T_1 \rightarrow T_2$ *arbitrary*: T_1 T_2 *rule*: *typing-induct*)
case *T-Abs*
show *?case* **by** *iprover*
next
case (*T-App* t_1 T_{11} t_2 T_1 T_2)
from ‹$t_1 \cdot t_2 \in value$›
show *?case* **by** *cases*
next
case (*T-TApp* t_1 T_{11} T_{12} T_2 T_1 $T_2{}'$)
from ‹$t_1 \cdot_\tau T_2 \in value$›
show *?case* **by** *cases*
next
case (*T-Sub* t S T_1 T_2)
from ‹$[] \vdash S <: T_1 \rightarrow T_2$›
obtain S_1 S_2 **where** S: $S = S_1 \rightarrow S_2$
by *cases* (*auto simp add*: *T-Sub*)
show *?case* **by** (*rule T-Sub S*)+
next
case (*T-Let* t_1 T_1 p Δ t_2 T_2 $T_1{}'$ $T_2{}'$)
from ‹(*LET* $p = t_1$ *IN* t_2) ∈ *value*›
show *?case* **by** *cases*
next

case ($T\text{-}Proj\ t\ fTs\ l\ T_1\ \ T_2$)
from ‹$t..l \in value$›
show *?case* **by** *cases*
qed *simp-all*

lemma *TyAll-canonical*: — A.14(3)
assumes *ty*: $[] \vdash v : (\forall <:T_1.\ T_2)$
shows $v \in value \Longrightarrow \exists\, t\ S.\ v = (\lambda{<:}S.\ t)$ **using** *ty*
proof ($induct\ []::env\ v\ \forall <:T_1.\ T_2\ arbitrary{:}\ T_1\ T_2\ rule{:}\ typing\text{-}induct$)
case ($T\text{-}App\ t_1\ \ T_{11}\ t_2\ \ T_1\ \ T_2$)
from ‹$t_1 \cdot t_2 \in value$›
show *?case* **by** *cases*
next
case *T-TAbs*
show *?case* **by** *iprover*
next
case ($T\text{-}TApp\ t_1\ \ T_{11}\ \ T_{12}\ \ T_2\ \ T_1\ \ T_2{}'$)
from ‹$t_1 \cdot_\tau\ T_2 \in value$›
show *?case* **by** *cases*
next
case ($T\text{-}Sub\ t\ S\ T_1\ \ T_2$)
from ‹$[] \vdash S <: (\forall <:T_1.\ T_2)$›
obtain $S_1\ S_2$ **where** S: $S = (\forall <:S_1.\ S_2)$
by *cases* (*auto simp add*: *T-Sub*)
show *?case* **by** (*rule T-Sub S*)+
next
case ($T\text{-}Let\ t_1\ \ T_1\ \ p\ \Delta\ t_2\ \ T_2\ \ T_1{}'\ T_2{}'$)
from ‹$(LET\ p = t_1\ IN\ t_2) \in value$›
show *?case* **by** *cases*
next
case ($T\text{-}Proj\ t\ fTs\ l\ T_1\ \ T_2$)
from ‹$t..l \in value$›
show *?case* **by** *cases*
qed *simp-all*

Like in the case of the simple calculus, we also need a canonical values theorem for record types:

lemma *RcdT-canonical*: — A.14(2)
assumes *ty*: $[] \vdash v : RcdT\ fTs$
shows $v \in value \Longrightarrow$
$\exists fs.\ v = Rcd\ fs \wedge (\forall (l,\ t) \in set\ fs.\ t \in value)$ **using** *ty*
proof ($induct\ []::env\ v\ RcdT\ fTs\ arbitrary{:}\ fTs\ rule{:}\ typing\text{-}induct$)
case ($T\text{-}App\ t_1\ \ T_{11}\ t_2\ fTs$)
from ‹$t_1 \cdot t_2 \in value$›
show *?case* **by** *cases*
next
case ($T\text{-}TApp\ t_1\ \ T_{11}\ \ T_{12}\ \ T_2\ fTs$)
from ‹$t_1 \cdot_\tau\ T_2 \in value$›
show *?case* **by** *cases*

next
  case (*T-Sub t S fTs*)
  from ‹$[] \vdash S <: RcdT\ fTs$›
  obtain fTs' **where** *S*: $S = RcdT\ fTs'$
    by *cases* (*auto simp add*: *T-Sub*)
  show *?case* **by** (*rule T-Sub S*)+
next
  case (*T-Let* t_1 T_1 *p* Δ t_2 T_2 *fTs*)
  from ‹$(LET\ p = t_1\ IN\ t_2) \in value$›
  show *?case* **by** *cases*
next
  case (*T-Rcd fs fTs*)
  from ‹$Rcd\ fs \in value$›
  show *?case* **using** *T-Rcd* **by** *cases simp-all*
next
  case (*T-Proj t fTs l fTs'*)
  from ‹$t..l \in value$›
  show *?case* **by** *cases*
qed *simp-all*

theorem *reorder-prop*:
  $\forall (l, t) \in set\ fs.\ P\ t \Longrightarrow \forall (l, U) \in set\ fTs.\ \exists u.\ fs\langle l\rangle_? = \lfloor u\rfloor \Longrightarrow$
    $\forall (l, t) \in set\ (map\ (\lambda(l, T).\ (l, the\ (fs\langle l\rangle_?)))\ fTs).\ P\ t$
proof (*induct fs*)
  case *Nil*
  then show *?case*
    by *auto*
next
  case (*Cons a fs*)
  then show *?case*
    by (*smt* (*verit*) *assoc-set case-prod-unfold imageE list.set-map option.collapse*
      option.simps(*3*))
qed

Another central property needed in the proof of the progress theorem is that well-typed matching is defined. This means that if the pattern *p* is compatible with the type *T* of the closed term *t* that it has to match, then it is always possible to extract a list of terms *ts* corresponding to the variables in *p*. Interestingly, this important property is missing in the description of the PoplMark Challenge [1].

theorem *ptyping-match*:
  $\vdash p : T \Rightarrow \Delta \Longrightarrow [] \vdash t : T \Longrightarrow t \in value \Longrightarrow$
    $\exists ts.\ \vdash p \rhd t \Rightarrow ts$
  $\vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow [] \vdash fs\ [:]\ fTs \Longrightarrow$
    $\forall (l, t) \in set\ fs.\ t \in value \Longrightarrow \exists us.\ \vdash fps\ [\rhd]\ fs \Rightarrow us$
proof (*induct arbitrary*: *t* **and** *fs set*: *ptyping ptypings*)
  case (*P-Var T t*)
  show *?case* **by** (*iprover intro*: *M-PVar*)
next

case (*P-Rcd fps fTs* Δ *t*)
then obtain *fs* **where**
t: $t = Rcd\ fs$ **and** *fs*: $\forall (l, t) \in set\ fs.\ t \in value$
by (*blast dest*: *RcdT-canonical*)
with *P-Rcd* **have** *fs'*: $[] \vdash Rcd\ fs : RcdT\ fTs$ **by** *simp*
hence $[] \vdash map\ (\lambda(l, T).\ (l, the\ (fs\langle l\rangle_?)))\ fTs\ [:]\ fTs$
by (*rule Rcd-type2'*)
moreover from *Rcd-type1'* [*OF fs'*]
have *assoc*: $\forall (l, U) \in set\ fTs.\ \exists u.\ fs\langle l\rangle_? = \lfloor u\rfloor$ **by** *blast*
with *fs* **have** $\forall (l, t) \in set\ (map\ (\lambda(l, T).\ (l, the\ (fs\langle l\rangle_?)))\ fTs).\ t \in value$
by (*rule reorder-prop*)
ultimately have $\exists us.\ \vdash fps\ [\rhd]\ map\ (\lambda(l, T).\ (l, the\ (fs\langle l\rangle_?)))\ fTs \Rightarrow us$
by (*rule P-Rcd*)
then obtain *us* **where** $\vdash fps\ [\rhd]\ map\ (\lambda(l, T).\ (l, the\ (fs\langle l\rangle_?)))\ fTs \Rightarrow us$ **..**
with *P-Rcd*(*1*) *assoc* **have** $\vdash fps\ [\rhd]\ fs \Rightarrow us$ **by** (*rule matchs-reorder'*)
hence $\vdash PRcd\ fps \rhd Rcd\ fs \Rightarrow us$ **by** (*rule M-Rcd*)
with *t* **show** *?case* **by** *fastforce*
next
case (*P-Nil fs*)
show *?case* **by** (*iprover intro*: *M-Nil*)
next
case (*P-Cons p T* Δ_1 *fps fTs* Δ_2 *l fs*)
from ‹$[] \vdash fs\ [:]\ (l, T) :: fTs$›
obtain *t fs'* **where** *fs*: $fs = (l, t) :: fs'$ **and** *t*: $[] \vdash t : T$
and *fs'*: $[] \vdash fs'\ [:]\ fTs$ **by** *cases auto*
have $((l, t) :: fs')\langle l\rangle_? = \lfloor t\rfloor$ **by** *simp*
moreover from *fs P-Cons* **have** $t \in value$ **by** *simp*
with *t* **have** $\exists ts.\ \vdash p \rhd t \Rightarrow ts$ **by** (*rule P-Cons*)
then obtain *ts* **where** $\vdash p \rhd t \Rightarrow ts$ **..**
moreover from *P-Cons fs* **have** $\forall (l, t) \in set\ fs'.\ t \in value$ **by** *auto*
with *fs'* **have** $\exists us.\ \vdash fps\ [\rhd]\ fs' \Rightarrow us$ **by** (*rule P-Cons*)
then obtain *us* **where** $\vdash fps\ [\rhd]\ fs' \Rightarrow us$ **..**
hence $\vdash fps\ [\rhd]\ (l, t) :: fs' \Rightarrow us$ **using** *P-Cons*(*5*) **by** (*rule matchs-mono*)
ultimately have $\vdash (l, p) :: fps\ [\rhd]\ (l, t) :: fs' \Rightarrow ts\ @\ us$
by (*rule M-Cons*)
with *fs* **show** *?case* **by** *iprover*
qed

theorem *progress*: — A.16
$[] \vdash t : T \Longrightarrow t \in value \lor (\exists t'.\ t \longmapsto t')$
$[] \vdash fs\ [:]\ fTs \Longrightarrow (\forall (l, t) \in set\ fs.\ t \in value) \lor (\exists fs'.\ fs\ [\longmapsto]\ fs')$
proof (*induct* []::*env t T* **and** []::*env fs fTs set*: *typing typings*)
case *T-Var*
thus *?case* **by** *simp*
next
case *T-Abs*
from *value.Abs* **show** *?case* **..**
next
case (*T-App* t_1 T_{11} T_{12} t_2)

hence $t_1 \in \mathit{value} \vee (\exists\, t'.\ t_1 \longmapsto t')$ **by** *simp*
thus *?case*
proof
assume t_1*-val*: $t_1 \in \mathit{value}$
with *T-App* **obtain** $t\ S$ **where** t_1: $t_1 = (\lambda{:}S.\ t)$
by (*auto dest*!: *Fun-canonical*)
from *T-App* **have** $t_2 \in \mathit{value} \vee (\exists\, t'.\ t_2 \longmapsto t')$ **by** *simp*
thus *?thesis*
proof
assume $t_2 \in \mathit{value}$
with t_1 **have** $t_1 \cdot t_2 \longmapsto t[0 \mapsto t_2]$
by *simp* (*rule eval-evals.intros*)
thus *?thesis* **by** *iprover*
next
assume $\exists\, t'.\ t_2 \longmapsto t'$
then obtain t' **where** $t_2 \longmapsto t'$ **by** *iprover*
with t_1*-val* **have** $t_1 \cdot t_2 \longmapsto t_1 \cdot t'$ **by** (*rule eval-evals.intros*)
thus *?thesis* **by** *iprover*
qed
next
assume $\exists\, t'.\ t_1 \longmapsto t'$
then obtain t' **where** $t_1 \longmapsto t'$ **..**
hence $t_1 \cdot t_2 \longmapsto t' \cdot t_2$ **by** (*rule eval-evals.intros*)
thus *?thesis* **by** *iprover*
qed
next
case *T-TAbs*
from *value.TAbs* **show** *?case* **..**
next
case (*T-TApp* $t_1\ T_{11}\ T_{12}\ T_2$)
hence $t_1 \in \mathit{value} \vee (\exists\, t'.\ t_1 \longmapsto t')$ **by** *simp*
thus *?case*
proof
assume $t_1 \in \mathit{value}$
with *T-TApp* **obtain** $t\ S$ **where** $t_1 = (\lambda{<}{:}S.\ t)$
by (*auto dest*!: *TyAll-canonical*)
hence $t_1 \cdot_\tau T_2 \longmapsto t[0 \mapsto_\tau T_2]$ **by** *simp* (*rule eval-evals.intros*)
thus *?thesis* **by** *iprover*
next
assume $\exists\, t'.\ t_1 \longmapsto t'$
then obtain t' **where** $t_1 \longmapsto t'$ **..**
hence $t_1 \cdot_\tau T_2 \longmapsto t' \cdot_\tau T_2$ **by** (*rule eval-evals.intros*)
thus *?thesis* **by** *iprover*
qed
next
case (*T-Sub* $t\ S\ T$)
show *?case* **by** (*rule T-Sub*)
next
case (*T-Let* $t_1\ T_1\ p\ \Delta\ t_2\ T_2$)

hence $t_1 \in \mathit{value} \vee (\exists\, t'.\ t_1 \longmapsto t')$ **by** *simp*
thus *?case*
proof
assume t_1: $t_1 \in \mathit{value}$
with *T-Let* **have** $\exists\, ts.\ \vdash p \rhd t_1 \Rightarrow ts$
by (*auto intro*: *ptyping-match*)
with t_1 **show** *?thesis* **by** (*blast intro*: *eval-evals.intros*)
next
assume $\exists\, t'.\ t_1 \longmapsto t'$
thus *?thesis* **by** (*blast intro*: *eval-evals.intros*)
qed
next
case (*T-Rcd fs fTs*)
thus *?case* **by** (*blast intro*: *value.intros eval-evals.intros*)
next
case (*T-Proj t fTs l T*)
hence $t \in \mathit{value} \vee (\exists\, t'.\ t \longmapsto t')$ **by** *simp*
thus *?case*
proof
assume *tv*: $t \in \mathit{value}$
with *T-Proj* **obtain** *fs* **where**
t: $t = \mathit{Rcd}\ \mathit{fs}$ **and** *fs*: $\forall\,(l,\ t) \in \mathit{set}\ \mathit{fs}.\ t \in \mathit{value}$
by (*auto dest*: *RcdT-canonical*)
with *T-Proj* **have** $[] \vdash \mathit{Rcd}\ \mathit{fs} : \mathit{RcdT}\ \mathit{fTs}$ **by** *simp*
hence $\forall\,(l,\ U) \in \mathit{set}\ \mathit{fTs}.\ \exists\, u.\ \mathit{fs}\langle l\rangle_? = \lfloor u \rfloor \wedge [] \vdash u : U$
by (*rule Rcd-type1'*)
with *T-Proj* **obtain** *u* **where** *u*: $\mathit{fs}\langle l\rangle_? = \lfloor u \rfloor$ **by** (*blast dest*: *assoc-set*)
with *fs* **have** $u \in \mathit{value}$ **by** (*blast dest*: *assoc-set*)
with *u t* **show** *?case* **by** (*blast intro*: *eval-evals.intros*)
next
assume $\exists\, t'.\ t \longmapsto t'$
thus *?case* **by** (*blast intro*: *eval-evals.intros*)
qed
next
case *T-Nil*
show *?case* **by** *simp*
next
case (*T-Cons t T fs fTs l*)
thus *?case* **by** (*auto intro*: *eval-evals.intros*)
qed

4 Evaluation contexts

In this section, we present a different way of formalizing the evaluation relation. Rather than using additional congruence rules, we first formalize a set *ctxt* of evaluation contexts, describing the locations in a term where reductions can occur. We have chosen a higher-order formalization of evaluation contexts as functions from terms to terms. We define simultaneously a set *rctxt* of evaluation contexts for records represented as functions from terms to lists of fields.

inductive-set
  $ctxt :: (trm \Rightarrow trm)\ set$
  and $rctxt :: (trm \Rightarrow rcd)\ set$
where
  $C\text{-}Hole\colon (\lambda t.\ t) \in ctxt$
| $C\text{-}App1\colon E \in ctxt \Longrightarrow (\lambda t.\ E\ t \cdot u) \in ctxt$
| $C\text{-}App2\colon v \in value \Longrightarrow E \in ctxt \Longrightarrow (\lambda t.\ v \cdot E\ t) \in ctxt$
| $C\text{-}TApp\colon E \in ctxt \Longrightarrow (\lambda t.\ E\ t \cdot_\tau\ T) \in ctxt$
| $C\text{-}Proj\colon E \in ctxt \Longrightarrow (\lambda t.\ E\ t..l) \in ctxt$
| $C\text{-}Rcd\colon E \in rctxt \Longrightarrow (\lambda t.\ Rcd\ (E\ t)) \in ctxt$
| $C\text{-}Let\colon E \in ctxt \Longrightarrow (\lambda t.\ LET\ p = E\ t\ IN\ u) \in ctxt$
| $C\text{-}hd\colon E \in ctxt \Longrightarrow (\lambda t.\ (l,\ E\ t) :: fs) \in rctxt$
| $C\text{-}tl\colon v \in value \Longrightarrow E \in rctxt \Longrightarrow (\lambda t.\ (l,\ v) :: E\ t) \in rctxt$

lemmas $rctxt\text{-}induct = ctxt\text{-}rctxt.inducts(2)$
  $[of\ \text{-}\ \lambda x.\ True,\ simplified\ True\text{-}simps,\ consumes\ 1,\ case\text{-}names\ C\text{-}hd\ C\text{-}tl]$

lemma *rctxt-labels*:
  assumes $H\colon E \in rctxt$
  shows $E\ t\langle l\rangle_? = \bot \Longrightarrow E\ t'\langle l\rangle_? = \bot$ **using** H
  by (*induct rule*: *rctxt-induct*) *auto*

The evaluation relation $t \longmapsto_c t'$ is now characterized by the rule *E-Ctxt*, which allows reductions in arbitrary contexts, as well as the rules *E-Abs*, *E-TAbs*, *E-LetV*, and *E-ProjRcd* describing the "immediate" reductions, which have already been presented in §2.6 and §3.6.

inductive
  $eval :: trm \Rightarrow trm \Rightarrow bool$ (**infixl** $\langle\longmapsto_c\rangle$ 50)
where
  $E\text{-}Ctxt\colon t \longmapsto_c t' \Longrightarrow E \in ctxt \Longrightarrow E\ t \longmapsto_c E\ t'$
| $E\text{-}Abs\colon v_2 \in value \Longrightarrow (\lambda{:}T_{11}.\ t_{12}) \cdot v_2 \longmapsto_c t_{12}[0 \mapsto v_2]$
| $E\text{-}TAbs\colon (\lambda{<:}T_{11}.\ t_{12}) \cdot_\tau\ T_2 \longmapsto_c t_{12}[0 \mapsto_\tau T_2]$
| $E\text{-}LetV\colon v \in value \Longrightarrow \vdash p \rhd v \Rightarrow ts \Longrightarrow (LET\ p = v\ IN\ t) \longmapsto_c t[0 \mapsto_s ts]$
| $E\text{-}ProjRcd\colon fs\langle l\rangle_? = \lfloor v\rfloor \Longrightarrow v \in value \Longrightarrow Rcd\ fs..l \longmapsto_c v$

In the proof of the preservation theorem, the case corresponding to the rule *E-Ctxt* requires a lemma stating that replacing a term t in a well-typed term of the form $E\ t$, where E is a context, by a term t' of the same type does not change the type of the resulting term $E\ t'$. The proof is by mutual induction on the typing derivations for terms and records.

lemma *context-typing*: — A.18
  $\Gamma \vdash u : T \Longrightarrow E \in ctxt \Longrightarrow u = E\ t \Longrightarrow$
    $(\bigwedge T_0.\ \Gamma \vdash t : T_0 \Longrightarrow \Gamma \vdash t' : T_0) \Longrightarrow \Gamma \vdash E\ t' : T$
  $\Gamma \vdash fs\ [:]\ fTs \Longrightarrow E_r \in rctxt \Longrightarrow fs = E_r\ t \Longrightarrow$

$(\bigwedge T_0.\ \Gamma \vdash t : T_0 \Longrightarrow \Gamma \vdash t' : T_0) \Longrightarrow \Gamma \vdash E_r\ t'\ [:]\ fTs$

proof (*induct arbitrary*: $E\ t\ t'$ **and** $E_r\ t\ t'$ *set*: *typing typings*)
  case (*T-Var* $\Gamma\ i\ U\ T\ E\ t\ t'$)
  from ‹$E \in ctxt$›
  have $E = (\lambda t.\ t)$ **using** *T-Var* **by** *cases simp-all*
  with *T-Var* **show** *?case* **by** (*blast intro*: *typing-typings.intros*)
next
  case (*T-Abs* $T_1\ T_2\ \Gamma\ t_2\ E\ t\ t'$)
  from ‹$E \in ctxt$›
  have $E = (\lambda t.\ t)$ **using** *T-Abs* **by** *cases simp-all*
  with *T-Abs* **show** *?case* **by** (*blast intro*: *typing-typings.intros*)
next
  case (*T-App* $\Gamma\ t_1\ T_{11}\ T_{12}\ t_2\ E\ t\ t'$)
  from ‹$E \in ctxt$›
  show *?case* **using** *T-App*
    by *cases* (*simp-all*, (*blast intro*: *typing-typings.intros*)+)
next
  case (*T-TAbs* $T_1\ \Gamma\ t_2\ T_2\ E\ t\ t'$)
  from ‹$E \in ctxt$›
  have $E = (\lambda t.\ t)$ **using** *T-TAbs* **by** *cases simp-all*
  with *T-TAbs* **show** *?case* **by** (*blast intro*: *typing-typings.intros*)
next
  case (*T-TApp* $\Gamma\ t_1\ T_{11}\ T_{12}\ T_2\ E\ t\ t'$)
  from ‹$E \in ctxt$›
  show *?case* **using** *T-TApp*
    by *cases* (*simp-all*, (*blast intro*: *typing-typings.intros*)+)
next
  case (*T-Sub* $\Gamma\ t\ S\ T\ E\ ta\ t'$)
  thus *?case* **by** (*blast intro*: *typing-typings.intros*)
next
  case (*T-Let* $\Gamma\ t_1\ T_1\ p\ \Delta\ t_2\ T_2\ E\ t\ t'$)
  from ‹$E \in ctxt$›
  show *?case* **using** *T-Let*
    by *cases* (*simp-all*, (*blast intro*: *typing-typings.intros*)+)
next
  case (*T-Rcd* Γ *fs fTs* $E\ t\ t'$)
  from ‹$E \in ctxt$›
  show *?case* **using** *T-Rcd*
    by *cases* (*simp-all*, (*blast intro*: *typing-typings.intros*)+)
next
  case (*T-Proj* $\Gamma\ t$ *fTs* $l\ T\ E\ ta\ t'$)
  from ‹$E \in ctxt$›
  show *?case* **using** *T-Proj*
    by *cases* (*simp-all*, (*blast intro*: *typing-typings.intros*)+)
next
  case (*T-Nil* $\Gamma\ E\ t\ t'$)
  from ‹$E \in rctxt$›
  show *?case* **using** *T-Nil*
    by *cases simp-all*

next
  case ($T\text{-}Cons\ \Gamma\ t\ T\ fs\ fTs\ l\ E\ ta\ t'$)
  from ‹$E \in rctxt$›
  show *?case* **using** *T-Cons*
    by *cases* (*blast intro*: *typing-typings.intros rctxt-labels*)+
qed

The fact that immediate reduction preserves the types of terms is proved in several parts. The proof of each statement is by induction on the typing derivation.

theorem *Abs-preservation*: — A.19(1)
  assumes H: $\Gamma \vdash (\lambda{:}T_{11}.\ t_{12}) \cdot t_2 : T$
  shows $\Gamma \vdash t_{12}[0 \mapsto t_2] : T$
  using H
proof (*induct* Γ $(\lambda{:}T_{11}.\ t_{12}) \cdot t_2$ T *arbitrary*: T_{11} t_{12} t_2 *rule*: *typing-induct*)
  case ($T\text{-}App\ \Gamma\ T_{11}\ T_{12}\ t_2\ T_{11}'\ t_{12}$)
  from ‹$\Gamma \vdash (\lambda{:}T_{11}'.\ t_{12}) : T_{11} \to T_{12}$›
  obtain S'
    where T_{11}: $\Gamma \vdash T_{11} <: T_{11}'$
    and t_{12}: $VarB\ T_{11}' :: \Gamma \vdash t_{12} : S'$
    and S': $\Gamma \vdash S'[0 \mapsto_\tau Top]_\tau <: T_{12}$ **by** (*rule Abs-type'* [*simplified*]) *blast*
  from ‹$\Gamma \vdash t_2 : T_{11}$›
  have $\Gamma \vdash t_2 : T_{11}'$ **using** T_{11} **by** (*rule T-Sub*)
  with t_{12} **have** $\Gamma \vdash t_{12}[0 \mapsto t_2] : S'[0 \mapsto_\tau Top]_\tau$
    by (*rule subst-type* [**where** Δ=[], *simplified*])
  then show *?case* **using** S' **by** (*rule T-Sub*)
next
  case *T-Sub*
  thus *?case* **by** (*blast intro*: *typing-typings.intros*)
qed

theorem *TAbs-preservation*: — A.19(2)
  assumes H: $\Gamma \vdash (\lambda{<:}T_{11}.\ t_{12}) \cdot_\tau T_2 : T$
  shows $\Gamma \vdash t_{12}[0 \mapsto_\tau T_2] : T$
  using H
proof (*induct* Γ $(\lambda{<:}T_{11}.\ t_{12}) \cdot_\tau T_2$ T *arbitrary*: T_{11} t_{12} T_2 *rule*: *typing-induct*)
  case ($T\text{-}TApp\ \Gamma\ T_{11}\ T_{12}\ T_2\ T_{11}'\ t_{12}$)
  from ‹$\Gamma \vdash (\lambda{<:}T_{11}'.\ t_{12}) : (\forall{<:}T_{11}.\ T_{12})$›
  obtain S'
    where $TVarB\ T_{11} :: \Gamma \vdash t_{12} : S'$
    and $TVarB\ T_{11} :: \Gamma \vdash S' <: T_{12}$ **by** (*rule TAbs-type'*) *blast*
  hence $TVarB\ T_{11} :: \Gamma \vdash t_{12} : T_{12}$ **by** (*rule T-Sub*)
  then show *?case* **using** ‹$\Gamma \vdash T_2 <: T_{11}$›
    by (*rule substT-type* [**where** Δ=[], *simplified*])
next
  case *T-Sub*
  thus *?case* **by** (*blast intro*: *typing-typings.intros*)
qed

theorem *Let-preservation*: — A.19(3)
  assumes H: $\Gamma \vdash (LET\ p = t_1\ IN\ t_2) : T$
  shows $\vdash p \rhd t_1 \Rightarrow ts \Longrightarrow \Gamma \vdash t_2[0 \mapsto_s ts] : T$
  using H
proof (*induct* Γ $LET\ p = t_1\ IN\ t_2$ T *arbitrary*: $p\ t_1\ t_2\ ts$ *rule*: *typing-induct*)
  case ($T\text{-}Let\ \Gamma\ t_1\ T_1\ p\ \Delta\ t_2\ T_2\ ts$)
  from ‹$\vdash p : T_1 \Rightarrow \Delta$› ‹$\Gamma \vdash t_1 : T_1$› ‹$\Delta @ \Gamma \vdash t_2 : T_2$› ‹$\vdash p \rhd t_1 \Rightarrow ts$›
  show *?case*
    by (*rule match-type*(*1*) [*of* - - - - - [], *simplified*])
next
  case *T-Sub*
  thus *?case* **by** (*blast intro*: *typing-typings.intros*)
qed

theorem *Proj-preservation*: — A.19(4)
  assumes H: $\Gamma \vdash Rcd\ fs..l : T$
  shows $fs\langle l\rangle_? = \lfloor v\rfloor \Longrightarrow \Gamma \vdash v : T$
  using H
proof (*induct* Γ *Rcd fs..l T arbitrary*: *fs l v rule*: *typing-induct*)
  case (*T-Proj* Γ *fTs l T fs v*)
  from ‹$\Gamma \vdash Rcd\ fs : RcdT\ fTs$›
  have $\forall (l,\ U)\in set\ fTs.\ \exists u.\ fs\langle l\rangle_? = \lfloor u\rfloor \wedge \Gamma \vdash u : U$
    by (*rule Rcd-type1'*)
  with *T-Proj* **show** *?case* **by** (*fastforce dest*: *assoc-set*)
next
  case *T-Sub*
  thus *?case* **by** (*blast intro*: *typing-typings.intros*)
qed

theorem *preservation*: — A.20
  assumes H: $t \longmapsto_c t'$
  shows $\Gamma \vdash t : T \Longrightarrow \Gamma \vdash t' : T$ **using** H
proof (*induct arbitrary*: Γ T)
  case (*E-Ctxt* $t\ t'\ E\ \Gamma\ T$)
  from *E-Ctxt*(*4*,*3*) *refl E-Ctxt*(*2*)
  show *?case* **by** (*rule context-typing*)
next
  case (*E-Abs* $v_2\ T_{11}\ t_{12}\ \Gamma\ T$)
  from *E-Abs*(*2*)
  show *?case* **by** (*rule Abs-preservation*)
next
  case (*E-TAbs* $T_{11}\ t_{12}\ T_2\ \Gamma\ T$)
  thus *?case* **by** (*rule TAbs-preservation*)
next
  case (*E-LetV* $v\ p\ ts\ t\ \Gamma\ T$)
  from *E-LetV*(*3*,*2*)
  show *?case* **by** (*rule Let-preservation*)
next
  case (*E-ProjRcd* $fs\ l\ v\ \Gamma\ T$)

from *E-ProjRcd*(*3*,*1*)
show *?case* **by** (*rule Proj-preservation*)
qed

For the proof of the progress theorem, we need a lemma stating that each well-typed, closed term t is either a canonical value, or can be decomposed into an evaluation context E and a term t_0 such that t_0 is a redex. The proof of this result, which is called the *decomposition lemma*, is again by induction on the typing derivation. A similar property is also needed for records.

theorem *context-decomp*: — A.15
$[] \vdash t : T \Longrightarrow$
$t \in value \vee (\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t = E\ t_0 \wedge t_0 \longmapsto_c t_0')$
$[] \vdash fs\ [:]\ fTs \Longrightarrow$
$(\forall (l, t) \in set\ fs.\ t \in value) \vee (\exists E\ t_0\ t_0'.\ E \in rctxt \wedge fs = E\ t_0 \wedge t_0 \longmapsto_c t_0')$
proof (*induct* []::*env t T* **and** []::*env fs fTs set*: *typing typings*)
case *T-Var*
thus *?case* **by** *simp*
next
case *T-Abs*
from *value.Abs* **show** *?case* **..**
next
case (*T-App* t_1 T_{11} T_{12} t_2)
from ‹$t_1 \in value \vee (\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t_1 = E\ t_0 \wedge t_0 \longmapsto_c t_0')$›
show *?case*
proof
assume t_1-*val*: $t_1 \in value$
with *T-App* **obtain** *t S* **where** t_1: $t_1 = (\lambda{:}S.\ t)$
by (*auto dest*!: *Fun-canonical*)
from ‹$t_2 \in value \vee (\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t_2 = E\ t_0 \wedge t_0 \longmapsto_c t_0')$›
show *?thesis*
proof
assume $t_2 \in value$
with t_1 **have** $t_1 \cdot t_2 \longmapsto_c t[0 \mapsto t_2]$
by *simp* (*rule eval.intros*)
thus *?thesis* **by** (*iprover intro*: *C-Hole*)
next
assume $\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t_2 = E\ t_0 \wedge t_0 \longmapsto_c t_0'$
with t_1-*val* **show** *?thesis* **by** (*iprover intro*: *ctxt-rctxt.intros*)
qed
next
assume $\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t_1 = E\ t_0 \wedge t_0 \longmapsto_c t_0'$
thus *?thesis* **by** (*iprover intro*: *ctxt-rctxt.intros*)
qed
next
case *T-TAbs*
from *value.TAbs* **show** *?case* **..**
next
case (*T-TApp* t_1 T_{11} T_{12} T_2)

from ‹$t_1 \in value \vee (\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t_1 = E\ t_0 \wedge t_0 \longmapsto_c t_0')$›
show *?case*
proof
assume $t_1 \in value$
with *T-TApp* **obtain** $t\ S$ **where** $t_1 = (\lambda{<}{:}S.\ t)$
by (*auto dest*!: *TyAll-canonical*)
hence $t_1 \cdot_\tau T_2 \longmapsto_c t[0 \mapsto_\tau T_2]$ **by** *simp* (*rule eval.intros*)
thus *?thesis* **by** (*iprover intro*: *C-Hole*)
next
assume $\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t_1 = E\ t_0 \wedge t_0 \longmapsto_c t_0'$
thus *?thesis* **by** (*iprover intro*: *ctxt-rctxt.intros*)
qed
next
case (*T-Sub t S T*)
show *?case* **by** (*rule T-Sub*)
next
case (*T-Let* $t_1\ T_1\ p\ \Delta\ t_2\ T_2$)
from ‹$t_1 \in value \vee (\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t_1 = E\ t_0 \wedge t_0 \longmapsto_c t_0')$›
show *?case*
proof
assume t_1: $t_1 \in value$
with *T-Let* **have** $\exists ts.\ \vdash p \rhd t_1 \Rightarrow ts$
by (*auto intro*: *ptyping-match*)
with t_1 **show** *?thesis* **by** (*iprover intro*: *eval.intros C-Hole*)
next
assume $\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t_1 = E\ t_0 \wedge t_0 \longmapsto_c t_0'$
thus *?thesis* **by** (*iprover intro*: *ctxt-rctxt.intros*)
qed
next
case (*T-Rcd fs fTs*)
thus *?case* **by** (*blast intro*: *value.intros eval.intros ctxt-rctxt.intros*)
next
case (*T-Proj t fTs l T*)
from ‹$t \in value \vee (\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t = E\ t_0 \wedge t_0 \longmapsto_c t_0')$›
show *?case*
proof
assume *tv*: $t \in value$
with *T-Proj* **obtain** *fs* **where**
t: $t = Rcd\ fs$ **and** *fs*: $\forall (l, t) \in set\ fs.\ t \in value$
by (*auto dest*: *RcdT-canonical*)
with *T-Proj* **have** $[] \vdash Rcd\ fs : RcdT\ fTs$ **by** *simp*
hence $\forall (l, U) {\in} set\ fTs.\ \exists u.\ fs\langle l\rangle_? = \lfloor u \rfloor \wedge [] \vdash u : U$
by (*rule Rcd-type1'*)
with *T-Proj* **obtain** *u* **where** *u*: $fs\langle l\rangle_? = \lfloor u \rfloor$ **by** (*blast dest*: *assoc-set*)
with *fs* **have** $u \in value$ **by** (*blast dest*: *assoc-set*)
with *u t* **show** *?thesis* **by** (*iprover intro*: *eval.intros C-Hole*)
next
assume $\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t = E\ t_0 \wedge t_0 \longmapsto_c t_0'$
thus *?case* **by** (*iprover intro*: *ctxt-rctxt.intros*)

qed
next
case *T-Nil*
show *?case* **by** *simp*
next
case (*T-Cons t T fs fTs l*)
thus *?case* **by** (*auto intro*: *ctxt-rctxt.intros*)
qed

theorem *progress*: — A.16
assumes *H*: $[] \vdash t : T$
shows $t \in value \vee (\exists t'.\ t \longmapsto_c t')$
proof −
from *H* **have** $t \in value \vee (\exists E\ t_0\ t_0{}'.\ E \in ctxt \wedge t = E\ t_0 \wedge t_0 \longmapsto_c t_0{}')$
by (*rule context-decomp*)
thus *?thesis* **by** (*iprover intro*: *eval.intros*)
qed

Finally, we prove that the two definitions of the evaluation relation are equivalent. The proof that $t \longmapsto_c t'$ implies $t \longmapsto t'$ requires a lemma stating that $\longmapsto$ is compatible with evaluation contexts.

lemma *ctxt-imp-eval*:
$E \in ctxt \Longrightarrow t \longmapsto t' \Longrightarrow E\ t \longmapsto E\ t'$
$E_r \in rctxt \Longrightarrow t \longmapsto t' \Longrightarrow E_r\ t\ [\longmapsto]\ E_r\ t'$
by (*induct rule*: *ctxt-rctxt.inducts*) (*auto intro*: *eval-evals.intros*)

lemma *eval-evalc-eq*: $(t \longmapsto t') = (t \longmapsto_c t')$
proof
fix *ts ts'*
have *r*: $t \longmapsto t' \Longrightarrow t \longmapsto_c t'$ **and**
$ts\ [\longmapsto]\ ts' \Longrightarrow \exists E\ t\ t'.\ E \in rctxt \wedge ts = E\ t \wedge ts' = E\ t' \wedge t \longmapsto_c t'$
by (*induct rule*: *eval-evals.inducts*) (*iprover intro*: *ctxt-rctxt.intros eval.intros*)+
assume $t \longmapsto t'$
thus $t \longmapsto_c t'$ **by** (*rule r*)
next
assume $t \longmapsto_c t'$
thus $t \longmapsto t'$
by *induct* (*auto intro*: *eval-evals.intros ctxt-imp-eval*)
qed

5 Executing the specification

An important criterion that a solution to the PoplMark Challenge should fulfill is the possibility to *animate* the specification. For example, it should be possible to apply the reduction relation for the calculus to example terms. Since the reduction relations are defined inductively, they can be interpreted as a logic program in the style of Prolog. The definition of the single-step evaluation relation presented in §2.6 and §3.6 is directly executable.

In order to compute the normal form of a term using the one-step evaluation relation $\longmapsto$, we introduce the inductive predicate $t \Downarrow u$, denoting that u is a normal form of t.

inductive *norm* :: *trm* ⇒ *trm* ⇒ *bool* (**infixl** ‹⇓› *50*)
where
  $t \in value \Longrightarrow t \Downarrow t$
| $t \longmapsto s \Longrightarrow s \Downarrow u \Longrightarrow t \Downarrow u$

definition *normal-forms* **where**
  normal-forms $t \equiv \{u.\ t \Downarrow u\}$

lemma [*code-pred-intro Rcd-Nil*]: *valuep* (*Rcd* [])
by (*auto intro*: *valuep.intros*)

lemma [*code-pred-intro Rcd-Cons*]: *valuep t* ⟹ *valuep* (*Rcd fs*) ⟹ *valuep* (*Rcd* ((*l*, *t*) # *fs*))
by (*auto intro*!: *valuep.intros elim*!: *valuep.cases*)

lemmas *valuep.intros*(*1*)[*code-pred-intro Abs'*] *valuep.intros*(*2*)[*code-pred-intro TAbs'*]

code-pred (*modes*: *i* => *bool*) *valuep*
proof −
  case *valuep*
  from *valuep.prems* **show** *thesis*
  proof (*cases rule*: *valuep.cases*)
    case (*Rcd fs*)
    from *this valuep.Rcd-Nil valuep.Rcd-Cons* **show** *thesis*
      by (*cases fs*) (*auto intro*: *valuep.intros*)
  next
    case *Abs*
    with *valuep.Abs'* **show** *thesis* **.**
  next
    case *TAbs*
    with *valuep.TAbs'* **show** *thesis* **.**
  qed
qed

thm *valuep.equation*

code-pred (*modes*: *i* => *i* => *bool*, *i* => *o* => *bool as normalize*) *norm* **.**

thm *norm.equation*

lemma [*code*]:
  normal-forms = *set-of-pred o normalize*
unfolding *set-of-pred-def o-def normal-forms-def* [*abs-def*]

by (*auto intro*: *set-eqI normalizeI elim*: *normalizeE*)

lemma [*code-unfold*]: $x \in value \longleftrightarrow valuep\ x$
by (*simp add*: *value-def*)

definition
natT :: *type* **where**
$natT \equiv \forall<:Top.\ (\forall<:TVar\ 0.\ (\forall<:TVar\ 1.\ (TVar\ 2 \to TVar\ 1) \to TVar\ 0 \to TVar\ 1))$

definition
fact2 :: *trm* **where**
$fact2 \equiv$
LET PVar natT =
$(\lambda<:Top.\ \lambda<:TVar\ 0.\ \lambda<:TVar\ 1.\ \lambda:TVar\ 2 \to TVar\ 1.\ \lambda:\ TVar\ 1.\ Var\ 1 \cdot Var\ 0)$
IN
LET PRcd
$[(''pluspp'',\ PVar\ (natT \to natT \to natT)),$
$(''multpp'',\ PVar\ (natT \to natT \to natT))] = Rcd$
$[(''multpp'',\ \lambda:natT.\ \lambda:natT.\ \lambda<:Top.\ \lambda<:TVar\ 0.\ \lambda<:TVar\ 1.\ \lambda:TVar\ 2 \to TVar\ 1.$
$Var\ 5 \cdot_\tau TVar\ 3 \cdot_\tau TVar\ 2 \cdot_\tau TVar\ 1 \cdot (Var\ 4 \cdot_\tau TVar\ 3 \cdot_\tau TVar\ 2 \cdot_\tau TVar\ 1) \cdot Var\ 0),$
$(''pluspp'',\ \lambda:natT.\ \lambda:natT.\ \lambda<:Top.\ \lambda<:TVar\ 0.\ \lambda<:TVar\ 1.\ \lambda:TVar\ 2 \to TVar\ 1.\ \lambda:TVar\ 1.$
$Var\ 6 \cdot_\tau TVar\ 4 \cdot_\tau TVar\ 3 \cdot_\tau TVar\ 3 \cdot Var\ 1 \cdot$
$(Var\ 5 \cdot_\tau TVar\ 4 \cdot_\tau TVar\ 3 \cdot_\tau TVar\ 2 \cdot Var\ 1 \cdot Var\ 0))]$
IN
$Var\ 0 \cdot (Var\ 1 \cdot Var\ 2 \cdot Var\ 2) \cdot Var\ 2$

value *normal-forms fact2*

Unfortunately, the definition based on evaluation contexts from §4 is not directly executable. The reason is that from the definition of evaluation contexts, the code generator cannot immediately read off an algorithm that, given a term t, computes a context E and a term t_0 such that $t = E\ t_0$. In order to do this, one would have to extract the algorithm contained in the proof of the *decomposition lemma* from §4.

values $\{u.\ norm\ fact2\ u\}$

References

[1] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The POPLMARK Challenge. In

T. Melham and J. Hurd, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2005*, LNCS. Springer-Verlag, 2005.

[2] B. Barras and B. Werner. Coq in Coq. To appear in Journal of Automated Reasoning.

[3] T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66, 2001.