# A Solution to the PoplMark Challenge in Isabelle/HOL

Stefan Berghofer

February 23, 2021

**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*
**apply** −
**apply** *rule*
**apply** (*erule meta-mp*)
**apply** (*rule TrueI*)
**apply** *assumption*
**apply** *rule*
**apply** (*rule TrueI*)
**apply** *assumption*
**apply** *rule*
**apply** (*rule TrueI*)+
**done**

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* ($\lfloor$-$\rfloor$)

**notation**
*None* ($\bot$)

**notation**
*length* ($\|$-$\|$)

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

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

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

**lemma** [*simp*]: *i* < $\|$*xs*$\|$ ⟹ (*xs* @ *ys*)⟨*i*⟩ = *xs*⟨*i*⟩
**apply** (*induct xs arbitrary*: *i*)
**apply** *simp*
**apply** (*case-tac i*)
**apply** *simp-all*
**done**

**lemma** [*simp*]: $\|$*xs*$\|$ ≤ *i* ⟹ (*xs* @ *ys*)⟨*i*⟩ = *ys*⟨*i* − $\|$*xs*$\|$⟩
**apply** (*induct xs arbitrary*: *i*)
**apply** *simp*
**apply** (*case-tac i*)
**apply** *simp-all*
**done**

Association lists

**primrec** *assoc* :: (*'a* × *'b*) *list* ⇒ *'a* ⇒ *'b option* (-⟨-⟩$_?$ [*90*, *0*] *91*)
**where**
[]⟨*a*⟩$_?$ = $\bot$
| (*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*⟩$_?$ = $\bot$ ∧ *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*⟩$_?$ = $\bot$ ⟹ *map* (λ(*x*, *y*). (*x*, *f x y*)) *ps*⟨*x*⟩$_?$ = $\bot$
**by** (*induct ps*) *auto*

**no-syntax**
*-Map* :: *maplets* => *'a* ⇀ *'b* ((*1*[-]))

# 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 $i$th 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* ((*3*λ:-./ -) [*0*, *10*] *10*)
| *TAbs type trm* ((*3*λ<:-./ -) [*0*, *10*] *10*)
| *App trm trm* (**infixl** · *200*)
| *TApp trm type* (**infixl** ·$_\tau$ *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 ↑$_\tau$ *n k* and ↑ *n k* for types and terms, respectively, which increment the indices of all free variables with indices ≥ *k* by *n*. The lifting functions on types and terms are defined by

**primrec** *liftT* :: *nat* ⇒ *nat* ⇒ *type* ⇒ *type* (↑$_\tau$)
**where**
  ↑$_\tau$ *n k* (*TVar i*) = (*if i* < *k then TVar i else TVar* (*i* + *n*))
| ↑$_\tau$ *n k Top* = *Top*
| ↑$_\tau$ *n k* (*T* → *U*) = ↑$_\tau$ *n k T* → ↑$_\tau$ *n k U*
| ↑$_\tau$ *n k* (∀<:*T*. *U*) = (∀<:↑$_\tau$ *n k T*. ↑$_\tau$ *n* (*k* + *1*) *U*)

**primrec** *lift* :: *nat* ⇒ *nat* ⇒ *trm* ⇒ *trm* (↑)
**where**
  ↑ *n k* (*Var i*) = (*if i* < *k then Var i else Var* (*i* + *n*))
| ↑ *n k* (λ:*T*. *t*) = (λ:↑$_\tau$ *n k T*. ↑ *n* (*k* + *1*) *t*)
| ↑ *n k* (λ<:*T*. *t*) = (λ<:↑$_\tau$ *n k T*. ↑ *n* (*k* + *1*) *t*)
| ↑ *n k* (*s* · *t*) = ↑ *n k s* · ↑ *n k t*
| ↑ *n k* (*t* ·$_\tau$ *T*) = ↑ *n k t* ·$_\tau$ ↑$_\tau$ *n k T*

It is useful to also define an "unlifting" function ↓$_\tau$ *n k* for decrementing all free variables with indices ≥ *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* (-[- ↦$_\tau$ -]$_\tau$ [*300*, *0*, *0*] *300*)
**where**
  (*TVar i*)[*k* ↦$_\tau$ *S*]$_\tau$ =
    (*if k* < *i then TVar* (*i* − *1*) *else if i* = *k then* ↑$_\tau$ *k 0 S else TVar i*)
| *Top*[*k* ↦$_\tau$ *S*]$_\tau$ = *Top*
| (*T* → *U*)[*k* ↦$_\tau$ *S*]$_\tau$ = *T*[*k* ↦$_\tau$ *S*]$_\tau$ → *U*[*k* ↦$_\tau$ *S*]$_\tau$
| (∀<:*T*. *U*)[*k* ↦$_\tau$ *S*]$_\tau$ = (∀<:*T*[*k* ↦$_\tau$ *S*]$_\tau$. *U*[*k*+*1* ↦$_\tau$ *S*]$_\tau$)

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

**primrec** *subst* :: *trm* ⇒ *nat* ⇒ *trm* ⇒ *trm* (-[- ↦ -] [*300*, *0*, *0*] *300*)
**where**
  (*Var i*)[*k* ↦ *s*] = (*if k* < *i then Var* (*i* − *1*) *else if i* = *k then* ↑ *k 0 s else Var i*)
| (*t* · *u*)[*k* ↦ *s*] = *t*[*k* ↦ *s*] · *u*[*k* ↦ *s*]
| (*t* ·$_\tau$ *T*)[*k* ↦ *s*] = *t*[*k* ↦ *s*] ·$_\tau$ ↓$_\tau$ *1 k T*
| (λ:*T*. *t*)[*k* ↦ *s*] = (λ:↓$_\tau$ *1 k T*. *t*[*k*+*1* ↦ *s*])
| (λ<:*T*. *t*)[*k* ↦ *s*] = (λ<:↓$_\tau$ *1 k T*. *t*[*k*+*1* ↦ *s*])

**primrec** *substT* :: *trm* ⇒ *nat* ⇒ *type* ⇒ *trm* (-[- ↦$_\tau$ -] [*300*, *0*, *0*] *300*)
**where**
  (*Var i*)[*k* ↦$_\tau$ *S*] = (*if k* < *i then Var* (*i* − *1*) *else Var i*)
| (*t* · *u*)[*k* ↦$_\tau$ *S*] = *t*[*k* ↦$_\tau$ *S*] · *u*[*k* ↦$_\tau$ *S*]
| (*t* ·$_\tau$ *T*)[*k* ↦$_\tau$ *S*] = *t*[*k* ↦$_\tau$ *S*] ·$_\tau$ *T*[*k* ↦$_\tau$ *S*]$_\tau$
| (λ:*T*. *t*)[*k* ↦$_\tau$ *S*] = (λ:*T*[*k* ↦$_\tau$ *S*]$_\tau$. *t*[*k*+*1* ↦$_\tau$ *S*])
| (λ<:*T*. *t*)[*k* ↦$_\tau$ *S*] = (λ<:*T*[*k* ↦$_\tau$ *S*]$_\tau$. *t*[*k*+*1* ↦$_\tau$ *S*])

Lifting and substitution extends to typing contexts as follows:

**primrec** *liftE* :: *nat* ⇒ *nat* ⇒ *env* ⇒ *env* (↑$_e$)
**where**
  ↑$_e$ *n k* [] = []
| ↑$_e$ *n k* (*B* :: Γ) = *mapB* (↑$_\tau$ *n* (*k* + ‖Γ‖)) *B* :: ↑$_e$ *n k* Γ

**primrec** *substE* :: *env* ⇒ *nat* ⇒ *type* ⇒ *env* (-[- ↦$_\tau$ -]$_e$ [*300*, *0*, *0*] *300*)
**where**
  []*k* ↦$_\tau$ *T*]$_e$ = []
| (*B* :: Γ)[*k* ↦$_\tau$ *T*]$_e$ = *mapB* (λ*U*. *U*[*k* + ‖Γ‖ ↦$_\tau$ *T*]$_\tau$) *B* :: Γ[*k* ↦$_\tau$ *T*]$_e$

**primrec** *decE* :: *nat* ⇒ *nat* ⇒ *env* ⇒ *env* (↓$_e$)
**where**
  ↓$_e$ *0 k* Γ = Γ
| ↓$_e$ (*Suc n*) *k* Γ = ↓$_e$ *n k* (Γ[*k* ↦$_\tau$ *Top*]$_e$)

Note that in a context of the form *B* :: Γ, 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 ‖Γ‖ 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*]: ‖↑$_e$ *n k* Γ‖ = ‖Γ‖
  **by** (*induct* Γ) *simp-all*

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

**lemma** *liftE-nth* [*simp*]:
$(\uparrow_e n\ k\ \Gamma)\langle i\rangle = \mathit{map\text{-}option}\ (\mathit{mapB}\ (\uparrow_\tau n\ (k + \|\Gamma\| - i - 1)))\ (\Gamma\langle i\rangle)$
**apply** (*induct* $\Gamma$ *arbitrary*: $i$)
**apply** *simp*
**apply** *simp*
**apply** (*case-tac* $i$)
**apply** *simp*
**apply** *simp*
**done**

**lemma** *substE-nth* [*simp*]:
$(\Gamma[0 \mapsto_\tau T]_e)\langle i\rangle = \mathit{map\text{-}option}\ (\mathit{mapB}\ (\lambda U.\ U[\|\Gamma\| - i - 1 \mapsto_\tau T]_\tau))\ (\Gamma\langle i\rangle)$
**apply** (*induct* $\Gamma$ *arbitrary*: $i$)
**apply** *simp*
**apply** *simp*
**apply** (*case-tac* $i$)
**apply** *simp*
**apply** *simp*
**done**

**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*]:
$i + m \le j \Longrightarrow \uparrow_\tau n\ j\ (\uparrow_\tau m\ i\ T) = \uparrow_\tau m\ i\ (\uparrow_\tau n\ (j - m)\ T)$
**apply** (*induct* $T$ *arbitrary*: $i\ j\ m\ n$)
**apply** *simp-all*
**apply** *arith*
**apply** (*subgoal-tac Suc* $j - m = \mathit{Suc}\ (j - m)$)
**apply** *simp*
**apply** *arith*
**done**

**lemma** *lift-size* [*simp*]: *size* $(\uparrow_\tau n\ k\ T) = \mathit{size}\ T$
**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*]:
$k \leq k' \Longrightarrow \uparrow_\tau\ n\ k\ (T[k' \mapsto_\tau\ U]_\tau) = \uparrow_\tau\ n\ k\ T[k' + n \mapsto_\tau\ U]_\tau$
**apply** (*induct T arbitrary*: $k\ k'$)
**apply** *simp-all*
**done**

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

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

**lemma** *liftT-substT-strange*:
$\uparrow_\tau\ n\ k\ T[n + k \mapsto_\tau\ U]_\tau = \uparrow_\tau\ n\ (Suc\ k)\ T[k \mapsto_\tau \uparrow_\tau\ n\ 0\ U]_\tau$
**apply** (*induct T arbitrary*: $n\ k$)
**apply** *simp-all*
**apply** (*thin-tac* $\bigwedge x.\ PROP\ P\ x$ **for** $P :: \text{-} \Rightarrow prop$)
**apply** (*drule-tac* $x{=}n$ **in** *meta-spec*)
**apply** (*drule-tac* $x{=}Suc\ k$ **in** *meta-spec*)
**apply** *simp*
**done**

**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*:
$i \leq j \Longrightarrow T[Suc\ j \mapsto_\tau\ V]_\tau[i \mapsto_\tau\ U[j - i \mapsto_\tau\ V]_\tau]_\tau = T[i \mapsto_\tau\ U]_\tau[j \mapsto_\tau\ V]_\tau$
**apply** (*induct T arbitrary*: $i\ j\ U\ V$)
**apply** (*simp-all add*: *diff-Suc split*: *nat.split*)
**apply** (*thin-tac* $\bigwedge x.\ PROP\ P\ x$ **for** $P :: \text{-} \Rightarrow prop$)
**apply** (*drule-tac* $x{=}Suc\ i$ **in** *meta-spec*)
**apply** (*drule-tac* $x{=}Suc\ j$ **in** *meta-spec*)
**apply** *simp*
**done**

## 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 $\Gamma$, if all variables occurring in it are defined in $\Gamma$. More precisely, if $T$ contains a type variable $TVar\ i$, then the $i$th element of $\Gamma$ must exist and have the form $TVarB\ U$.

**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*: $\Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Gamma \vdash_{wf} T \to U$
| *wf-all*: $\Gamma \vdash_{wf} T \Longrightarrow TVarB\ T :: \Gamma \vdash_{wf} U \Longrightarrow \Gamma \vdash_{wf} (\forall <: T.\ U)$

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

**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*: $\Gamma \vdash_{wfB} B \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow B :: \Gamma \vdash_{wf}$

The judgement $\Gamma \vdash_{wfB} B$, which denotes well-formedness of the binding $B$ with respect to context $\Gamma$, is just an abbreviation for $\Gamma \vdash_{wf} type\text{-}ofB\ B$. 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*:
  $\Gamma \vdash_{wf} TVar\ i$
  $\Gamma \vdash_{wf} Top$
  $\Gamma \vdash_{wf} T \to U$
  $\Gamma \vdash_{wf} (\forall <: T.\ U)$

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

**lemma** *wf-TVarB*: $\Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow TVarB\ T :: \Gamma \vdash_{wf}$
  **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* $\Gamma' =$ *map is-TVarB* $\Gamma \Longrightarrow$
    $\Gamma\langle i\rangle = \lfloor TVarB\ T\rfloor \Longrightarrow \exists\, T.\ \Gamma'\langle i\rangle = \lfloor TVarB\ T\rfloor$
  **apply** (*induct* $\Gamma$ *arbitrary*: $\Gamma'$ *T i*)

**apply** *simp*
**apply** (*auto split*: *nat.split-asm*)
**apply** (*case-tac z*)
**apply** *simp-all*
**done**

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

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

A well-formed context of the form $\Delta$ @ $B$ :: $\Gamma$ remains well-formed if we replace the binding $B$ by another well-formed binding $B'$:

**lemma** *wfE-replace*:
$\Delta$ @ $B$ :: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf\,B}$ $B' \Longrightarrow$ *is-TVarB* $B'$ = *is-TVarB* $B \Longrightarrow$
$\Delta$ @ $B'$ :: $\Gamma \vdash_{wf}$
**apply** (*induct* $\Delta$)
**apply** *simp*
**apply** (*erule wf-Cons*)
**apply** (*erule well-formedE-cases*)
**apply** *assumption*
**apply** *simp*
**apply** (*erule well-formedE-cases*)
**apply** (*rule wf-Cons*)
**apply** (*case-tac a*)
**apply** *simp*
**apply** (*rule wf-equallength*)
**apply** *assumption*
**apply** *simp*
**apply** *simp*
**apply** (*rule wf-equallength*)
**apply** *assumption*
**apply** *simp*
**apply** *simp*
**done**

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

**lemma** *wf-weaken*:
**assumes** *H*: $\Delta$ @ $\Gamma \vdash_{wf} T$
**shows** $\uparrow_e$ (*Suc 0*) *0* $\Delta$ @ $B$ :: $\Gamma \vdash_{wf} \uparrow_\tau$ (*Suc 0*) $\|\Delta\|$ $T$
**using** *H*
**apply** (*induct* $\Delta$ @ $\Gamma$ $T$ *arbitrary*: $\Delta$)
**apply** *simp-all*
**apply** (*rule conjI*)
**apply** (*rule impI*)

**apply** (*rule wf-TVar*)
**apply** *simp*
**apply** (*rule impI*)
**apply** (*rule wf-TVar*)
**apply** (*subgoal-tac Suc i − ‖Δ‖ = Suc (i − ‖Δ‖)*)
**apply** *simp*
**apply** *arith*
**apply** (*rule wf-Top*)
**apply** (*rule wf-arrow*)
**apply** *simp*
**apply** *simp*
**apply** (*rule wf-all*)
**apply** *simp*
**apply** *simp*
**done**

**lemma** *wf-weaken′*: $\Gamma \vdash_{wf} T \Longrightarrow \Delta \mathbin{@} \Gamma \vdash_{wf} \uparrow_\tau \|\Delta\|\ 0\ T$
**apply** (*induct* Δ)
**apply** *simp-all*
**apply** (*drule-tac B=a* **in** *wf-weaken* [*of* [], *simplified*])
**apply** *simp*
**done**

**lemma** *wfE-weaken*: $\Delta \mathbin{@} \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wfB} B \Longrightarrow \uparrow_e (Suc\ 0)\ 0\ \Delta \mathbin{@} B :: \Gamma \vdash_{wf}$
**apply** (*induct* Δ)
**apply** *simp*
**apply** (*rule wf-Cons*)
**apply** *assumption+*
**apply** *simp*
**apply** (*rule wf-Cons*)
**apply** (*erule well-formedE-cases*)
**apply** (*case-tac a*)
**apply** *simp*
**apply** (*rule wf-weaken*)
**apply** *assumption*
**apply** *simp*
**apply** (*rule wf-weaken*)
**apply** *assumption*
**apply** (*erule well-formedE-cases*)
**apply** *simp*
**done**

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 $\Gamma$ is valid in the full environment, as opposed to the smaller environment consisting only of the entries in $\Gamma$ 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*
**apply** (*induct arbitrary*: *i*)
**apply** *simp*
**apply** (*simp split*: *nat.split-asm*)
**apply** (*frule-tac B*=*VarB T* **in** *wf-weaken* [*of* [], *simplified*])
**apply** *simp*+
**apply** (*rename-tac nat*)
**apply** (*drule-tac x*=*nat* **in** *meta-spec*)
**apply** *simp*
**apply** (*frule-tac T*=$\uparrow_\tau$ (*Suc nat*) *0 T* **in** *wf-weaken* [*of* [], *simplified*])
**apply** *simp*
**done**

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

**theorem** *wf-subst*:
$\Delta$ @ $B$ :: $\Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Delta[0 \mapsto_\tau U]_e$ @ $\Gamma \vdash_{wf} T[\|\Delta\| \mapsto_\tau U]_\tau$
**apply** (*induct T arbitrary*: $\Delta$)
**apply** *simp-all*
**apply** (*rule conjI*)
**apply** (*rule impI*)
**apply** (*drule-tac* $\Gamma$=$\Gamma$ **and** $\Delta$=$\Delta[0 \mapsto_\tau U]_e$ **in** *wf-weaken'*)
**apply** *simp*
**apply** (*rule impI conjI*)+
**apply** (*erule well-formed-cases*)
**apply** (*rule wf-TVar*)
**apply** (*simp split*: *nat.split-asm*)
**apply** (*rename-tac nat* $\Delta$ *T nata*)
**apply** (*subgoal-tac* $\|\Delta\| \leq nat - Suc\ 0$)
**apply** (*subgoal-tac nat* $-$ *Suc* $\|\Delta\| = nata$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *arith*
**apply** (*rule impI*)
**apply** (*erule well-formed-cases*)
**apply** (*rule wf-TVar*)
**apply** *simp*
**apply** (*rule wf-Top*)
**apply** (*erule well-formed-cases*)
**apply** (*rule wf-arrow*)

**apply** *simp+*
**apply** (*erule well-formed-cases*)
**apply** (*rule wf-all*)
**apply** *simp*
**apply** (*thin-tac* $\bigwedge x.\ PROP\ P\ x$ **for** $P$ :: - $\Rightarrow$ *prop*)
**apply** (*drule-tac* $x$=*TVarB T1* :: $\Delta$ **in** *meta-spec*)
**apply** *simp*
**done**

**theorem** *wfE-subst*: $\Delta$ @ $B$ :: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Delta[0 \mapsto_\tau U]_e$ @ $\Gamma \vdash_{wf}$
**apply** (*induct* $\Delta$)
**apply** *simp*
**apply** (*erule well-formedE-cases*)
**apply** *assumption*
**apply** *simp*
**apply** (*case-tac a*)
**apply** (*erule well-formedE-cases*)
**apply** (*rule wf-Cons*)
**apply** *simp*
**apply** (*rule wf-subst*)
**apply** *assumption+*
**apply** *simp*
**apply** (*erule well-formedE-cases*)
**apply** (*rule wf-Cons*)
**apply** *simp*
**apply** (*rule wf-subst*)
**apply** *assumption+*
**done**

## 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* (- $\vdash$ - <: - [*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*: $\Gamma \vdash T_1 <: S_1 \Longrightarrow TVarB\ T_1 :: \Gamma \vdash S_2 <: T_2 \Longrightarrow$
$\Gamma \vdash (\forall <: S_1.\ S_2) <: (\forall <: T_1.\ T_2)$

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 $\Gamma$ must be of the form $\Gamma_1$ @ $B$ :: $\Gamma_2$, where $\Gamma_1$ has the length $i$. Since the indices of

variables in $B$ can only refer to variables defined in $\Gamma_2$, they have to be incremented by *Suc i* to ensure that they point to the right variables in the larger context $\Gamma$.

**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*
**apply** (*rule H'*)
**apply** (*rule wf-subtype-env*)
**apply** (*rule H*)
**apply** (*rule wf-subtype* [*OF H*, *THEN conjunct1*])
**apply** (*rule wf-subtype* [*OF H*, *THEN conjunct2*])
**done**

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*: $\Delta$)
**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 < ‖Δ‖)
    case True
    with SA-trans-TVar
    have (↑e 1 0 Δ @ B :: Γ)⟨i⟩ = ⌊TVarB (↑τ 1 (‖Δ‖ − Suc i) U)⌋
      by simp
    moreover from True SA-trans-TVar
    have ↑e 1 0 Δ @ B :: Γ ⊢
      ↑τ (Suc i) 0 (↑τ 1 (‖Δ‖ − Suc i) U) <: ↑τ 1 ‖Δ‖ T
      by simp
    ultimately have ↑e 1 0 Δ @ B :: Γ ⊢ TVar i <: ↑τ 1 ‖Δ‖ T
      by (rule subtyping.SA-trans-TVar)
    with True show ?thesis by simp
  next
    case False
    then have Suc i − ‖Δ‖ = Suc (i − ‖Δ‖) by arith
    with False SA-trans-TVar have (↑e 1 0 Δ @ B :: Γ)⟨Suc i⟩ = ⌊TVarB U⌋
      by simp
    moreover from False SA-trans-TVar
    have ↑e 1 0 Δ @ B :: Γ ⊢ ↑τ (Suc (Suc i)) 0 U <: ↑τ 1 ‖Δ‖ T
      by simp
    ultimately have ↑e 1 0 Δ @ B :: Γ ⊢ TVar (Suc i) <: ↑τ 1 ‖Δ‖ 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-all T1 S1 S2 T2 Δ)
  with SA-all(4) [of TVarB T1 :: Δ]
  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 $\Delta$ to a context $\Gamma$ preserves subtyping can be proved by induction on $\Delta$, using the previous result in the induction step:

```
lemma subtype-weaken': — A.2
  Γ ⊢ P <: Q ⟹ Δ @ Γ ⊢wf ⟹ Δ @ Γ ⊢ ↑τ ‖Δ‖ 0 P <: ↑τ ‖Δ‖ 0 Q
  apply (induct Δ)
  apply simp-all
  apply (erule well-formedE-cases)
```

**apply** *simp*
**apply** (*drule-tac B=a* **and** Γ=Δ @ Γ **in** *subtype-weaken* [*of* [], *simplified*])
**apply** *simp-all*
**done**

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 $\Delta$ @ *TVarB* $Q$ :: $\Gamma \vdash M <: N$. In the case corresponding to the rule *SA-trans-TVar*, we must prove $\Delta$ @ *TVarB* $P$ :: $\Gamma \vdash$ *TVar* $i <: T$. The only interesting case is the one where $i = \|\Delta\|$. By induction hypothesis, we know that $\Delta$ @ *TVarB* $P$ :: $\Gamma \vdash \uparrow_\tau (i + 1)\ 0\ Q <: T$ and $(\Delta$ @ *TVarB* $Q$ :: $\Gamma)\langle i\rangle = \lfloor$*TVarB* $Q\rfloor$. By assumption, we have $\Gamma \vdash P <: Q$ and hence $\Delta$ @ *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 $\Delta$ @ *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
$\Gamma \vdash S <: Q \Longrightarrow \Gamma \vdash Q <: T \Longrightarrow \Gamma \vdash S <: T$
$\Delta$ @ *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*: Γ *S T* Δ *P M N rule*: *wf-induct-rule*)
**case** (*less Q*)
**{**
**fix** Γ *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* Γ *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 T₁' T₂')
      from SA-arrow SA-arrow' have Γ ⊢ S₁ → S₂ <: T₁' → T₂'
        by (auto intro!: subtyping.SA-arrow intro: less(1) [of T₁] less(1) [of T₂])
      with SA-arrow show ?thesis by simp
    qed
  next
    case (SA-all Γ T₁ S₁ S₂ T₂)
    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 T₁' T₂')
      from SA-all SA-all' have Γ ⊢ T₁' <: S₁
        by − (rule less(1), simp-all)
      moreover from SA-all SA-all' have TVarB T₁' :: Γ ⊢ S₂ <: T₂
        by − (rule less(2) [of - []], simplified], simp-all)
      with SA-all SA-all' have TVarB T₁' :: Γ ⊢ S₂ <: T₂'
        by − (rule less(1), simp-all)
      ultimately have Γ ⊢ (∀ <:S₁. S₂) <: (∀ <:T₁'. T₂')
        by (rule subtyping.SA-all)
      with SA-all 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 :: Γ ⊢ 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 < ‖Δ‖)
        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 = ‖Δ‖)
          case True
          from SA-trans-TVar have (Δ @ [TVarB P]) @ Γ ⊢wf
            by (auto elim!: wf-subtypeE)
          with ‹Γ ⊢ P <: Q›
           have (Δ @ [TVarB P]) @ Γ ⊢ ↑τ ‖Δ @ [TVarB P]‖ 0 P <: ↑τ ‖Δ @
[TVarB P]‖ 0 Q
            by (rule subtype-weaken')
          with SA-trans-TVar True False have Δ @ TVarB P :: Γ ⊢ ↑τ (Suc ‖Δ‖)
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])
    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 :: 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
```

**apply** (*induct* Δ @ *TVarB Q* :: Γ *S T arbitrary*: Δ)
**apply** *simp-all*
**apply** (*rule SA-Top*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*rule wf-subst*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*rule impI conjI*)+
**apply** (*rule subtype-refl*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** (*drule-tac* $T{=}P$ **and** $\Delta{=}\Delta[0 \mapsto_\tau P]_e$ **in** *wf-weaken'*)
**apply** *simp*
**apply** (*rule conjI impI*)+
**apply** (*rule SA-refl-TVar*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** (*drule wf-subst*)
**apply** *assumption*
**apply** *simp*
**apply** (*rule impI*)
**apply** (*rule SA-refl-TVar*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** (*drule wf-subst*)
**apply** *assumption*
**apply** *simp*
**apply** (*rule conjI impI*)+
**apply** *simp*
**apply** (*drule-tac* $\Gamma{=}\Gamma$ **and** $\Delta{=}\Delta[0 \mapsto_\tau P]_e$ **in** *subtype-weaken'*)
**apply** (*erule wf-subtypeE*)+
**apply** *assumption*
**apply** *simp*
**apply** (*rule subtype-trans*(*1*))
**apply** *assumption*+
**apply** (*rule conjI impI*)+

**apply** (*rule SA-trans-TVar*)
**apply** (*simp split*: *nat.split-asm*)
**apply** (*subgoal-tac* $\|\Delta\| \leq i - Suc\ 0$)
**apply** (*rename-tac nat*)
**apply** (*subgoal-tac* $i - Suc\ \|\Delta\| = nat$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *arith*
**apply** *simp*
**apply** (*rule impI*)
**apply** (*rule SA-trans-TVar*)
**apply** (*simp split*: *nat.split-asm*)
**apply** (*subgoal-tac Suc* ($\|\Delta\| - Suc\ 0$) = $\|\Delta\|$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** (*rule SA-arrow*)
**apply** *simp*+
**apply** (*rule SA-all*)
**apply** *simp*
**apply** *simp*
**done**

**lemma** *subst-subtype*:
**assumes** *H*: $\Delta$ @ *VarB V* :: $\Gamma \vdash T <: U$
**shows** $\downarrow_e 1\ 0\ \Delta$ @ $\Gamma \vdash \downarrow_\tau 1\ \|\Delta\|\ T <: \downarrow_\tau 1\ \|\Delta\|\ U$
**using** *H*
**apply** (*induct* $\Delta$ @ *VarB V* :: $\Gamma$ *T U arbitrary*: $\Delta$)
**apply** *simp-all*
**apply** (*rule SA-Top*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)
**apply** (*rule wf-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)
**apply** (*rule impI conjI*)+
**apply** (*rule SA-Top*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)+
**apply** (*rule conjI impI*)+
**apply** (*rule SA-refl-TVar*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)
**apply** (*drule wf-subst*)
**apply** (*rule wf-Top*)
**apply** *simp*
**apply** (*rule impI*)

**apply** (*rule SA-refl-TVar*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)
**apply** (*drule wf-subst*)
**apply** (*rule wf-Top*)
**apply** *simp*
**apply** (*rule conjI impI*)+
**apply** *simp*
**apply** (*rule conjI impI*)+
**apply** (*simp split*: *nat.split-asm*)
**apply** (*rule SA-trans-TVar*)
**apply** (*subgoal-tac* $\|\Delta\| \leq i - Suc\ 0$)
**apply** (*rename-tac nat*)
**apply** (*subgoal-tac* $i - Suc\ \|\Delta\| = nat$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *arith*
**apply** *simp*
**apply** (*rule impI*)
**apply** (*rule SA-trans-TVar*)
**apply** *simp*
**apply** (*subgoal-tac* $0 < \|\Delta\|$)
**apply** *simp*
**apply** *arith*
**apply** (*rule SA-arrow*)
**apply** *simp*+
**apply** (*rule SA-all*)
**apply** *simp*
**apply** *simp*
**done**

## 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*: $\Gamma \vdash_{wf} \Longrightarrow \Gamma\langle i\rangle = \lfloor VarB\ U \rfloor \Longrightarrow T = \uparrow_\tau (Suc\ i)\ 0\ U \Longrightarrow \Gamma \vdash Var\ i : T$
| *T-Abs*: $VarB\ T_1 :: \Gamma \vdash t_2 : T_2 \Longrightarrow \Gamma \vdash (\lambda{:}T_1.\ t_2) : T_1 \to \downarrow_\tau 1\ 0\ T_2$
| *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*: $TVarB\ T_1 :: \Gamma \vdash t_2 : T_2 \Longrightarrow \Gamma \vdash (\lambda{<:}T_1.\ t_2) : (\forall{<:}T_1.\ T_2)$
| *T-TApp*: $\Gamma \vdash t_1 : (\forall{<:}T_{11}.\ T_{12}) \Longrightarrow \Gamma \vdash T_2 <: T_{11} \Longrightarrow$
    $\Gamma \vdash t_1 \cdot_\tau T_2 : T_{12}[0 \mapsto_\tau T_2]_\tau$
| *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 $\Gamma$ need to be incremented in order for the type to be well-formed with respect to $\Gamma$. 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*
**apply** *induct*
**apply** *simp*
**apply** (*rule wf-liftB*)
**apply** *assumption*+
**apply** (*drule wf-typeE1*)+
**apply** (*erule well-formedE-cases*)+
**apply** (*rule wf-arrow*)
**apply** *simp*
**apply** *simp*
**apply** (*rule wf-subst* [*of* [], *simplified*])
**apply** *assumption*
**apply** (*rule wf-Top*)
**apply** (*erule well-formed-cases*)
**apply** *assumption*
**apply** (*rule wf-all*)
**apply** (*drule wf-typeE1*)
**apply** (*erule well-formedE-cases*)
**apply** *simp*
**apply** *assumption*
**apply** (*erule well-formed-cases*)
**apply** (*rule wf-subst* [*of* [], *simplified*])
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**done**

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** $H$: $\Delta$ @ $TVarB\ Q :: \Gamma \vdash t : T$
**shows** $\Gamma \vdash P <: Q \Longrightarrow \Delta$ @ $TVarB\ P :: \Gamma \vdash t : T$
**using** $H$
**apply** (*induct* $\Delta$ @ $TVarB\ Q :: \Gamma\ t\ T$ *arbitrary*: $\Delta$)
**apply** *simp-all*
**apply** (*rule T-Var*)
**apply** (*erule wfE-replace*)
**apply** (*erule wf-subtypeE*)
**apply** *simp+*
**apply** (*case-tac* $i < \|\Delta\|$)
**apply** *simp*
**apply** (*case-tac* $i = \|\Delta\|$)
**apply** *simp*
**apply** (*simp split*: *nat.split nat.split-asm*)+
**apply** (*rule T-Abs* [*simplified*])
**apply** *simp*
**apply** (*rule-tac* $T_{11}$=$T_{11}$ **in** *T-App*)
**apply** *simp+*
**apply** (*rule T-TAbs*)
**apply** *simp*
**apply** (*rule-tac* $T_{11}$=$T_{11}$ **in** *T-TApp*)
**apply** *simp*
**apply** (*rule subtype-trans*(*2*))
**apply** *assumption+*
**apply** (*rule-tac* $S$=$S$ **in** *T-Sub*)
**apply** *simp*
**apply** (*rule subtype-trans*(*2*))
**apply** *assumption+*
**done**

**lemma** *subtype-refl'*:
**assumes** *t*: $\Gamma \vdash t : T$
**shows** $\Gamma \vdash T <: T$
**proof** (*rule subtype-refl*)
**from** *t* **show** $\Gamma \vdash_{wf}$ **by** (*rule wf-typeE1*)
**from** *t* **show** $\Gamma \vdash_{wf} T$ **by** (*rule wf-typeE2*)
**qed**

**lemma** *Abs-type*: — A.13(1)
**assumes** *H*: $\Gamma \vdash (\lambda{:}S.\ s) : T$
**shows** $\Gamma \vdash T <: U \to U' \Longrightarrow$
$(\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow VarB\ S :: \Gamma \vdash s : S' \Longrightarrow$
$\Gamma \vdash \downarrow_\tau\ 1\ 0\ S' <: U' \Longrightarrow P) \Longrightarrow P$
**using** *H*
**proof** (*induct* $\Gamma\ \lambda{:}S.\ s\ T$ *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 ‹VarB T₁ :: Γ ⊢ t₂ : T₂› 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 H subtype-refl' [OF H]
  by (rule Abs-type) (rule R)

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 T₁ Γ t₂ T₂)
  from ‹Γ ⊢ (∀<:T₁. T₂) <: (∀<:U. U')›
  obtain ty1: Γ ⊢ U <: T₁ and ty2: TVarB U :: Γ ⊢ T₂ <: U'
    by cases simp-all
  from ‹TVarB T₁ :: Γ ⊢ t₂ : T₂›
  have TVarB U :: Γ ⊢ t₂ : T₂ 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: Γ ⊢ (λ<:S. s) : (∀<:U. U')
  and R: ⋀S'. Γ ⊢ U <: S ⟹ TVarB U :: Γ ⊢ s : S' ⟹
    TVarB U :: Γ ⊢ S' <: U' ⟹ P
  shows P using H subtype-refl' [OF H]
  by (rule TAbs-type) (rule R)

lemma T-eq: Γ ⊢ t : T ⟹ T = T' ⟹ Γ ⊢ 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** $H$
**apply** (*induct* $\Delta$ @ $\Gamma$ $t$ $T$ *arbitrary*: $\Delta$)
**apply** *simp-all*
**apply** (*rule conjI*)
**apply** (*rule impI*)
**apply** (*rule T-Var*)
**apply** (*erule wfE-weaken*)
**apply** *simp+*
**apply** (*rule impI*)
**apply** (*rule T-Var*)
**apply** (*erule wfE-weaken*)
**apply** *assumption*
**apply** (*subgoal-tac Suc i* − $\|\Delta\|$ = *Suc* ($i$ − $\|\Delta\|$))
**apply** *simp*
**apply** *arith*
**apply** (*rule refl*)
**apply** (*rule T-Abs* [*THEN T-eq*])
**apply** *simp*
**apply** *simp*
**apply** (*rule-tac* $T_{11}$=$\uparrow_\tau$ (*Suc 0*) $\|\Delta\|$ $T_{11}$ **in** *T-App*)
**apply** *simp*
**apply** *simp*
**apply** (*rule T-TAbs*)
**apply** *simp*
**apply** (*erule-tac T-TApp* [*THEN T-eq*])
**apply** (*drule subtype-weaken*)
**apply** *simp+*
**apply** (*case-tac* $\Delta$)
**apply** (*simp add*: *liftT-substT-strange* [*of - 0*, *simplified*])+
**apply** (*rule-tac* $S$=$\uparrow_\tau$ (*Suc 0*) $\|\Delta\|$ $S$ **in** *T-Sub*)
**apply** *simp*
**apply** (*drule subtype-weaken*)
**apply** *simp+*
**done**

We can strengthen this result, so as to mean that concatenating a new context $\Delta$ to the context $\Gamma$ preserves typing:

**lemma** *type-weaken′*: — A.5(6)
$\Gamma \vdash t : T \Longrightarrow \Delta$ @ $\Gamma \vdash_{wf} \Longrightarrow \Delta$ @ $\Gamma \vdash \uparrow \|\Delta\|\ 0\ t : \uparrow_\tau \|\Delta\|\ 0\ T$
**apply** (*induct* $\Delta$)
**apply** *simp*
**apply** *simp*
**apply** (*erule well-formedE-cases*)
**apply** *simp*
**apply** (*drule-tac* $B$=$a$ **in** *type-weaken* [*of* [], *simplified*])

**apply** *simp+*
**done**

This property is proved by structural induction on the context $\Delta$, 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 $\Delta$ 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*: $\Delta$ @ *VarB U* :: $\Gamma \vdash t : T$
**shows** $\Gamma \vdash u : U \Longrightarrow$
$\downarrow_e$ *1 0* $\Delta$ @ $\Gamma \vdash t[\|\Delta\| \mapsto u] : \downarrow_\tau$ *1* $\|\Delta\|$ *T* **using** *H*
**apply** (*induct* $\Delta$ @ *VarB U* :: $\Gamma$ *t T arbitrary*: $\Delta$)
**apply** *simp*
**apply** (*rule conjI*)
**apply** (*rule impI*)
**apply** *simp*
**apply** (*drule-tac* $\Delta$=$\Delta[0 \mapsto_\tau Top]_e$ **in** *type-weaken'*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)
**apply** *simp*
**apply** (*rule impI conjI*)+
**apply** (*simp split*: *nat.split-asm*)
**apply** (*rule T-Var*)
**apply** (*erule wfE-subst*)
**apply** (*rule wf-Top*)
**apply** (*subgoal-tac* $\|\Delta\| \leq i - Suc\ 0$)
**apply** (*rename-tac nat*)
**apply** (*subgoal-tac i* $-$ *Suc* $\|\Delta\| = nat$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *arith*
**apply** *simp*
**apply** (*rule impI*)
**apply** (*rule T-Var*)
**apply** (*erule wfE-subst*)
**apply** (*rule wf-Top*)
**apply** *simp*
**apply** (*subgoal-tac Suc* ($\|\Delta\| - Suc\ 0$) = $\|\Delta\|$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *simp*
**apply** (*rule T-Abs* [*THEN T-eq*])
**apply** *simp*
**apply** (*simp add*: *substT-substT* [*symmetric*])

**apply** *simp*
**apply** (*rule-tac* $T_{11}{=}T_{11}[\|\Delta\| \mapsto_\tau \mathit{Top}]_\tau$ **in** *T-App*)
**apply** *simp+*
**apply** (*rule T-TAbs*)
**apply** *simp*
**apply** *simp*
**apply** (*rule T-TApp* [*THEN T-eq*])
**apply** *simp*
**apply** (*rule subst-subtype* [*simplified*])
**apply** *assumption*
**apply** (*simp add*: *substT-substT* [*symmetric*])
**apply** (*rule-tac* $S{=}S[\|\Delta\| \mapsto_\tau \mathit{Top}]_\tau$ **in** *T-Sub*)
**apply** *simp*
**apply** *simp*
**apply** (*rule subst-subtype* [*simplified*])
**apply** *assumption*
**done**

**theorem** *substT-type*: — A.11
**assumes** *H*: $\Delta$ @ *TVarB Q* :: $\Gamma \vdash t : T$
**shows** $\Gamma \vdash P <: Q \Longrightarrow$
$\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash t[\|\Delta\| \mapsto_\tau P] : T[\|\Delta\| \mapsto_\tau P]_\tau$ **using** *H*
**apply** (*induct* $\Delta$ @ *TVarB Q* :: $\Gamma$ *t T arbitrary*: $\Delta$)
**apply** *simp-all*
**apply** (*rule impI conjI*)+
**apply** *simp*
**apply** (*rule T-Var*)
**apply** (*erule wfE-subst*)
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*simp split*: *nat.split-asm*)
**apply** (*subgoal-tac* $\|\Delta\| \leq i - \mathit{Suc}\ 0$)
**apply** (*rename-tac nat*)
**apply** (*subgoal-tac* $i - \mathit{Suc}\ \|\Delta\| = \mathit{nat}$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *arith*
**apply** *simp*
**apply** (*rule impI*)
**apply** (*case-tac* $i = \|\Delta\|$)
**apply** *simp*
**apply** (*rule T-Var*)
**apply** (*erule wfE-subst*)
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** *simp*
**apply** (*subgoal-tac* $i < \|\Delta\|$)
**apply** (*subgoal-tac Suc* $(\|\Delta\| - \mathit{Suc}\ 0) = \|\Delta\|$)
**apply** (*simp* (*no-asm-simp*))

**apply** *arith*
**apply** *arith*
**apply** (*rule T-Abs* [*THEN T-eq*])
**apply** *simp*
**apply** (*simp add*: *substT-substT* [*symmetric*])
**apply** (*rule-tac* $T_{11}{=}T_{11}[\|\Delta\| \mapsto_\tau P]_\tau$ **in** *T-App*)
**apply** *simp+*
**apply** (*rule T-TAbs*)
**apply** *simp*
**apply** (*rule T-TApp* [*THEN T-eq*])
**apply** *simp*
**apply** (*rule substT-subtype*)
**apply** *assumption*
**apply** *assumption*
**apply** (*simp add*: *substT-substT* [*symmetric*])
**apply** (*rule-tac* $S{=}S[\|\Delta\| \mapsto_\tau P]_\tau$ **in** *T-Sub*)
**apply** *simp*
**apply** (*rule substT-subtype*)
**apply** *assumption*
**apply** *assumption*
**done**

## 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. $\beta$-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** $\longmapsto$ *50*)
**where**

*E-Abs*: $v_2 \in \mathit{value} \Longrightarrow (\lambda{:}T_{11}.\ t_{12}) \cdot v_2 \longmapsto t_{12}[0 \mapsto v_2]$
| *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*: $v \in \mathit{value} \Longrightarrow t \longmapsto t' \Longrightarrow v \cdot t \longmapsto v \cdot t'$
| *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* $\Gamma$ *i* *U* *T* $t'$)
  **from** ‹*Var i* $\longmapsto t'$›
  **show** *?case* **by** *cases*
**next**
  **case** (*T-Abs* $T_1$ $\Gamma$ $t_2$ $T_2$ $t'$)
  **from** ‹$(\lambda{:}T_1.\ t_2) \longmapsto t'$›
  **show** *?case* **by** *cases*
**next**
  **case** (*T-App* $\Gamma$ $t_1$ $T_{11}$ $T_{12}$ $t_2$ $t'$)
  **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** $S'$: $\Gamma \vdash S'[0 \mapsto_\tau \mathit{Top}]_\tau <: T_{12}$ **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** $\Gamma \vdash t_{12}[0 \mapsto t_2] : S'[0 \mapsto_\tau \mathit{Top}]_\tau$
      **by** (*rule subst-type* [**where** $\Delta$=[], *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 Γ ⊢ t'' · t₂ : T₁₂ using ‹Γ ⊢ t₂ : T₁₁›
      by (rule typing.T-App)
    with E-App1 show ?thesis by simp
  next
    case (E-App2 t'')
    from ‹t₂ ⟶ t''›
    have Γ ⊢ t'' : T₁₁ by (rule T-App)
    with T-App(1) have Γ ⊢ t₁ · t'' : T₁₂
      by (rule typing.T-App)
    with E-App2 show ?thesis by simp
  qed
next
  case (T-TAbs T₁ Γ t₂ T₂ t')
  from ‹(λ<:T₁. t₂) ⟶ t'›
  show ?case by cases
next
  case (T-TApp Γ t₁ T₁₁ T₁₂ T₂ t')
  from ‹t₁ ·τ T₂ ⟶ t'›
  show ?case
  proof cases
    case (E-TAbs T₁₁' t₁₂)
    with T-TApp have Γ ⊢ (λ<:T₁₁'. t₁₂) : (∀<:T₁₁. T₁₂) by simp
    then obtain S'
      where TVarB T₁₁ :: Γ ⊢ t₁₂ : S'
      and TVarB T₁₁ :: Γ ⊢ S' <: T₁₂ by (rule TAbs-type') blast
    hence TVarB T₁₁ :: Γ ⊢ t₁₂ : T₁₂ by (rule T-Sub)
    hence Γ ⊢ t₁₂[0 ↦τ T₂] : T₁₂[0 ↦τ T₂]τ using T-TApp(3)
      by (rule substT-type [where Δ=[], simplified])
    with E-TAbs show ?thesis by simp
  next
    case (E-TApp t'')
    from ‹t₁ ⟶ t''›
    have Γ ⊢ t'' : (∀<:T₁₁. T₁₂) by (rule T-TApp)
    hence Γ ⊢ t'' ·τ T₂ : T₁₂[0 ↦τ T₂]τ using ‹Γ ⊢ T₂ <: T₁₁›
      by (rule typing.T-TApp)
    with E-TApp show ?thesis by simp
  qed
next
  case (T-Sub Γ t S T t')
  from ‹t ⟶ t'›
  have Γ ⊢ t' : S by (rule T-Sub)
  then show ?case using ‹Γ ⊢ 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*: $[] \vdash t : T$
**shows** $t \in value \vee (\exists\, t'.\ t \longmapsto t')$ **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* $T_1$), …, ($l_n$, *PVar* $T_n$)] = *Rcd* [($l_1$, $v_1$), …, ($l_n$, $v_n$)] *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* ⇒ *nat* ⇒ *type* ⇒ *type* (↑$_\tau$)

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

**primrec** $liftp :: nat \Rightarrow nat \Rightarrow pat \Rightarrow pat\ (\uparrow_p)$
**and** $liftrp :: nat \Rightarrow nat \Rightarrow rpat \Rightarrow rpat\ (\uparrow_{rp})$
**and** $liftfp :: nat \Rightarrow nat \Rightarrow fpat \Rightarrow fpat\ (\uparrow_{fp})$
**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** $lift :: nat \Rightarrow nat \Rightarrow trm \Rightarrow trm\ (\uparrow)$
**and** $liftr :: nat \Rightarrow nat \Rightarrow rcd \Rightarrow rcd\ (\uparrow_r)$
**and** $liftf :: nat \Rightarrow nat \Rightarrow fld \Rightarrow fld\ (\uparrow_f)$
**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** $substTT :: type \Rightarrow nat \Rightarrow type \Rightarrow type\ \ (\text{-}[\text{-} \mapsto_\tau \text{-}]_\tau\ [300,\ 0,\ 0]\ 300)$
**and** $substrTT :: rcdT \Rightarrow nat \Rightarrow type \Rightarrow rcdT\ \ (\text{-}[\text{-} \mapsto_\tau \text{-}]_{r\tau}\ [300,\ 0,\ 0]\ 300)$
**and** $substfTT :: fldT \Rightarrow nat \Rightarrow type \Rightarrow fldT\ \ (\text{-}[\text{-} \mapsto_\tau \text{-}]_{f\tau}\ [300,\ 0,\ 0]\ 300)$
**where**
$(TVar\ i)[k \mapsto_\tau S]_\tau =$
$\quad (if\ k < i\ then\ TVar\ (i - 1)\ else\ if\ i = k\ then \uparrow_\tau\ k\ 0\ S\ else\ TVar\ i)$
$| Top[k \mapsto_\tau S]_\tau = Top$
$| (T \to U)[k \mapsto_\tau S]_\tau = T[k \mapsto_\tau S]_\tau \to U[k \mapsto_\tau S]_\tau$
$| (\forall <: T.\ U)[k \mapsto_\tau S]_\tau = (\forall <: T[k \mapsto_\tau S]_\tau.\ U[k+1 \mapsto_\tau S]_\tau)$
$| (RcdT\ fs)[k \mapsto_\tau S]_\tau = RcdT\ (fs[k \mapsto_\tau S]_{r\tau})$
$| []\,[k \mapsto_\tau S]_{r\tau} = []$

$|\ (f :: fs)[k \mapsto_\tau S]_{r\tau} = f[k \mapsto_\tau S]_{f\tau} :: fs[k \mapsto_\tau S]_{r\tau}$
$|\ (l,\ T)[k \mapsto_\tau S]_{f\tau} = (l,\ T[k \mapsto_\tau S]_\tau)$

**primrec** *substpT* :: *pat* ⇒ *nat* ⇒ *type* ⇒ *pat* (-[- ↦$_\tau$ -]$_p$ [*300*, *0*, *0*] *300*)
**and** *substrpT* :: *rpat* ⇒ *nat* ⇒ *type* ⇒ *rpat* (-[- ↦$_\tau$ -]$_{rp}$ [*300*, *0*, *0*] *300*)
**and** *substfpT* :: *fpat* ⇒ *nat* ⇒ *type* ⇒ *fpat* (-[- ↦$_\tau$ -]$_{fp}$ [*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* ⇒ *nat* ⇒ *pat* ⇒ *pat* ($\downarrow_p$)
**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* ⇒ *nat* ⇒ *trm* ⇒ *trm* (-[- ↦ -] [*300*, *0*, *0*] *300*)
**and** *substr* :: *rcd* ⇒ *nat* ⇒ *trm* ⇒ *rcd* (-[- ↦ -]$_r$ [*300*, *0*, *0*] *300*)
**and** *substf* :: *fld* ⇒ *nat* ⇒ *trm* ⇒ *fld* (-[- ↦ -]$_f$ [*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* ⇒ *nat* ⇒ *type* ⇒ *trm* (-[- ↦$_\tau$ -] [*300*, *0*, *0*] *300*)
**and** *substrT* :: *rcd* ⇒ *nat* ⇒ *type* ⇒ *rcd* (-[- ↦$_\tau$ -]$_r$ [*300*, *0*, *0*] *300*)

```
  and substfT :: fld ⇒ nat ⇒ type ⇒ fld  (-[- ↦τ -]f [300, 0, 0] 300)
where
  (Var i)[k ↦τ S] = (if k < i then Var (i − 1) else Var i)
| (t · u)[k ↦τ S] = t[k ↦τ S] · u[k ↦τ S]
| (t ·τ T)[k ↦τ S] = t[k ↦τ S] ·τ T[k ↦τ S]τ
| (λ:T. t)[k ↦τ S] = (λ:T[k ↦τ S]τ. t[k+1 ↦τ S])
| (λ<:T. t)[k ↦τ S] = (λ<:T[k ↦τ S]τ. t[k+1 ↦τ S])
| (Rcd fs)[k ↦τ S] = Rcd (fs[k ↦τ S]r)
| (t..a)[k ↦τ S] = (t[k ↦τ S])..a
| (LET p = t IN u)[k ↦τ S] =
    (LET p[k ↦τ S]p = t[k ↦τ S] IN u[k + ‖p‖p ↦τ S])
| []r[k ↦τ S] = []
| (f :: fs)[k ↦τ S]r = f[k ↦τ S]f :: fs[k ↦τ S]r
| (l, t)[k ↦τ S]f = (l, t[k ↦τ S])

primrec liftE :: nat ⇒ nat ⇒ env ⇒ env (↑e)
where
  ↑e n k [] = []
| ↑e n k (B :: Γ) = mapB (↑τ n (k + ‖Γ‖)) B :: ↑e n k Γ

primrec substE :: env ⇒ nat ⇒ type ⇒ env  (-[- ↦τ -]e [300, 0, 0] 300)
where
  []e[k ↦τ T] = []
| (B :: Γ)e[k ↦τ T] = mapB (λU. U[k + ‖Γ‖ ↦τ T]τ) B :: Γ[k ↦τ 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 ⇒ nat ⇒ trm list ⇒ trm  (-[- ↦s -] [300, 0, 0] 300)
where
  t[k ↦s []] = t
| t[k ↦s u :: us] = t[k + ‖us‖ ↦ u][k ↦s us]

primrec decT :: nat ⇒ nat ⇒ type ⇒ type  (↓τ)
where
  ↓τ 0 k T = T
| ↓τ (Suc n) k T = ↓τ n k (T[k ↦τ Top]τ)

primrec decE :: nat ⇒ nat ⇒ env ⇒ env  (↓e)
where
  ↓e 0 k Γ = Γ
| ↓e (Suc n) k Γ = ↓e n k (Γ[k ↦τ Top]e)

primrec decrT :: nat ⇒ nat ⇒ rcdT ⇒ rcdT  (↓rτ)
where
  ↓rτ 0 k fTs = fTs
| ↓rτ (Suc n) k fTs = ↓rτ n k (fTs[k ↦τ Top]rτ)
```

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* $\Gamma$) *simp-all*

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

**lemma** *liftE-nth* [*simp*]:
$(\uparrow_e\ n\ k\ \Gamma)\langle i\rangle = \mathit{map\text{-}option}\ (\mathit{mapB}\ (\uparrow_\tau\ n\ (k + \|\Gamma\| - i - 1)))\ (\Gamma\langle i\rangle)$
**apply** (*induct* $\Gamma$ *arbitrary*: *i*)
**apply** *simp*
**apply** *simp*
**apply** (*case-tac i*)
**apply** *simp*
**apply** *simp*
**done**

**lemma** *substE-nth* [*simp*]:
$(\Gamma[0 \mapsto_\tau T]_e)\langle i\rangle = \mathit{map\text{-}option}\ (\mathit{mapB}\ (\lambda U.\ U[\|\Gamma\| - i - 1 \mapsto_\tau T]_\tau))\ (\Gamma\langle i\rangle)$
**apply** (*induct* $\Gamma$ *arbitrary*: *i*)
**apply** *simp*
**apply** *simp*
**apply** (*case-tac i*)
**apply** *simp*
**apply** *simp*
**done**

**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*]:
$i + m \le j \Longrightarrow \uparrow_\tau\ n\ j\ (\uparrow_\tau\ m\ i\ T) = \uparrow_\tau\ m\ i\ (\uparrow_\tau\ n\ (j - m)\ T)$
$i + m \le j \Longrightarrow \uparrow_{r\tau}\ n\ j\ (\uparrow_{r\tau}\ m\ i\ rT) = \uparrow_{r\tau}\ m\ i\ (\uparrow_{r\tau}\ n\ (j - m)\ rT)$
$i + m \le j \Longrightarrow \uparrow_{f\tau}\ n\ j\ (\uparrow_{f\tau}\ m\ i\ fT) = \uparrow_{f\tau}\ m\ i\ (\uparrow_{f\tau}\ n\ (j - m)\ fT)$
**apply** (*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*)
**apply** *simp-all*
**apply** *arith*
**apply** (*subgoal-tac Suc j − m = Suc* $(j - m)$)
**apply** *simp*
**apply** *arith*
**done**

**lemma** *lift-size* [*simp*]:

*size* ($\uparrow_\tau$ *n k T*) = *size T*
*size-list* (*size-prod* ($\lambda x.$ *0*) *size*) ($\uparrow_{r\tau}$ *n k rT*) = *size-list* (*size-prod* ($\lambda x.$ *0*) *size*) *rT*
*size-prod* ($\lambda x.$ *0*) *size* ($\uparrow_{f\tau}$ *n k fT*) = *size-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*]$_{f\tau}$
**apply** (*induct T* **and** *rT* **and** *fT arbitrary*: *k k'* **and** *k k'* **and** *k k'*
*rule*: *liftT.induct liftrT.induct liftfT.induct*)
**apply** *simp-all*
**done**

**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*]$_{r\tau}$

$k' < k \Longrightarrow$
$\uparrow_{fT}\ n\ k\ (fT[k' \mapsto_T U]_{fT}) = \uparrow_{fT}\ n\ (k + 1)\ fT[k' \mapsto_T \uparrow_T\ n\ (k - k')\ U]_{fT}$
**apply** (*induct* $T$ **and** $rT$ **and** $fT$ *arbitrary*: $k\ k'$ **and** $k\ k'$ **and** $k\ k'$
*rule*: *liftT.induct liftrT.induct liftfT.induct*)
**apply** *simp-all*
**apply** *arith*
**done**

**lemma** *liftT-substT-Top* [*simp*]:
$k \leq k' \Longrightarrow \uparrow_T\ n\ k'\ (T[k \mapsto_T Top]_T) = \uparrow_T\ n\ (Suc\ k')\ T[k \mapsto_T Top]_T$
$k \leq k' \Longrightarrow \uparrow_{rT}\ n\ k'\ (rT[k \mapsto_T Top]_{rT}) = \uparrow_{rT}\ n\ (Suc\ k')\ rT[k \mapsto_T Top]_{rT}$
$k \leq k' \Longrightarrow \uparrow_{fT}\ n\ k'\ (fT[k \mapsto_T Top]_{fT}) = \uparrow_{fT}\ n\ (Suc\ k')\ fT[k \mapsto_T Top]_{fT}$
**apply** (*induct* $T$ **and** $rT$ **and** $fT$ *arbitrary*: $k\ k'$ **and** $k\ k'$ **and** $k\ k'$
*rule*: *liftT.induct liftrT.induct liftfT.induct*)
**apply** *simp-all*
**apply** *arith*
**done**

**theorem** *liftE-substE* [*simp*]:
$k \leq k' \Longrightarrow \uparrow_e\ n\ k\ (\Gamma[k' \mapsto_T U]_e) = \uparrow_e\ n\ k\ \Gamma[k' + n \mapsto_T U]_e$
**apply** (*induct* $\Gamma$ *arbitrary*: $k\ k'$ **and** $k\ k'$ **and** $k\ k'$)
**apply** *simp-all*
**apply** (*case-tac a*)
**apply** (*simp-all add*: *ac-simps*)
**done**

**lemma** *liftT-decT* [*simp*]:
$k \leq k' \Longrightarrow \uparrow_T\ n\ k'\ (\downarrow_T\ m\ k\ T) = \downarrow_T\ m\ k\ (\uparrow_T\ n\ (m + k')\ T)$
**by** (*induct m arbitrary*: $T$) *simp-all*

**lemma** *liftT-substT-strange*:
$\uparrow_T\ n\ k\ T[n + k \mapsto_T U]_T = \uparrow_T\ n\ (Suc\ k)\ T[k \mapsto_T \uparrow_T\ n\ 0\ U]_T$
$\uparrow_{rT}\ n\ k\ rT[n + k \mapsto_T U]_{rT} = \uparrow_{rT}\ n\ (Suc\ k)\ rT[k \mapsto_T \uparrow_T\ n\ 0\ U]_{rT}$
$\uparrow_{fT}\ n\ k\ fT[n + k \mapsto_T U]_{fT} = \uparrow_{fT}\ n\ (Suc\ k)\ fT[k \mapsto_T \uparrow_T\ n\ 0\ U]_{fT}$
**apply** (*induct* $T$ **and** $rT$ **and** $fT$ *arbitrary*: $n\ k$ **and** $n\ k$ **and** $n\ k$
*rule*: *liftT.induct liftrT.induct liftfT.induct*)
**apply** *simp-all*
**apply** (*thin-tac* $\bigwedge x.\ PROP\ P\ x$ **for** $P$ :: - $\Rightarrow$ *prop*)
**apply** (*drule-tac* $x{=}n$ **in** *meta-spec*)
**apply** (*drule-tac* $x{=}Suc\ k$ **in** *meta-spec*)
**apply** *simp*
**done**

**lemma** *liftp-liftp* [*simp*]:
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_p\ n'\ k'\ (\uparrow_p\ n\ k\ p) = \uparrow_p\ (n + n')\ k\ p$
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_{rp}\ n'\ k'\ (\uparrow_{rp}\ n\ k\ rp) = \uparrow_{rp}\ (n + n')\ k\ rp$
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_{fp}\ n'\ k'\ (\uparrow_{fp}\ n\ k\ fp) = \uparrow_{fp}\ (n + n')\ k\ fp$
**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*]:
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow n'\ k'\ (\uparrow n\ k\ t) = \uparrow (n + n')\ k\ t$
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_r n'\ k'\ (\uparrow_r n\ k\ fs) = \uparrow_r (n + n')\ k\ fs$
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_f n'\ k'\ (\uparrow_f n\ k\ f) = \uparrow_f (n + n')\ k\ f$
**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*]:
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_e n'\ k'\ (\uparrow_e n\ k\ \Gamma) = \uparrow_e (n + n')\ k\ \Gamma$
**apply** (*induct* $\Gamma$ *arbitrary*: *k k'*)
**apply** *simp-all*
**apply** (*case-tac a*)
**apply** *simp-all*
**done**

**lemma** *liftE-liftE'* [*simp*]:
$i + m \leq j \Longrightarrow \uparrow_e n\ j\ (\uparrow_e m\ i\ \Gamma) = \uparrow_e m\ i\ (\uparrow_e n\ (j - m)\ \Gamma)$
**apply** (*induct* $\Gamma$ *arbitrary*: *i j m n*)
**apply** *simp-all*
**apply** (*case-tac a*)
**apply** *simp-all*
**done**

**lemma** *substT-substT*:
$i \leq j \Longrightarrow$
$T[Suc\ j \mapsto_\tau V]_\tau[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_\tau = T[i \mapsto_\tau U]_\tau[j \mapsto_\tau V]_\tau$
$i \leq j \Longrightarrow$
$rT[Suc\ j \mapsto_\tau V]_{r\tau}[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_{r\tau} = rT[i \mapsto_\tau U]_{r\tau}[j \mapsto_\tau V]_{r\tau}$
$i \leq j \Longrightarrow$
$fT[Suc\ j \mapsto_\tau V]_{f\tau}[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_{f\tau} = fT[i \mapsto_\tau U]_{f\tau}[j \mapsto_\tau V]_{f\tau}$
**apply** (*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*)
**apply** (*simp-all add*: *diff-Suc split*: *nat.split*)
**apply** (*thin-tac* $\bigwedge x.$ *PROP P x* **for** *P* :: - $\Rightarrow$ *prop*)
**apply** (*drule-tac x*=*Suc i* **in** *meta-spec*)
**apply** (*drule-tac x*=*Suc j* **in** *meta-spec*)
**apply** *simp*
**done**

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

**lemma** *substT-decT'* [*simp*]:
$i \leq j \Longrightarrow \downarrow_\tau k\ (Suc\ j)\ T[i \mapsto_\tau Top]_\tau = \downarrow_\tau k\ j\ (T[i \mapsto_\tau Top]_\tau)$
**by** (*induct k arbitrary*: *i j T*) (*simp-all add*: *substT-substT* [*of - - - - Top*, *simplified*])

**lemma** *substE-substE*:
$i \leq j \Longrightarrow \Gamma[Suc\ j \mapsto_\tau V]_e[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_e = \Gamma[i \mapsto_\tau U]_e[j \mapsto_\tau V]_e$
**apply** (*induct* $\Gamma$)
**apply** (*case-tac* [*2*] *a*)
**apply** (*simp-all add*: *substT-substT* [*symmetric*])
**done**

**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* $\Gamma$) (*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* $\Gamma$ *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* $\Gamma$) (*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*]:
$\downarrow_e n\ k\ (B :: \Gamma) = mapB\ (\downarrow_\tau n\ (k + \|\Gamma\|))\ B :: \downarrow_e n\ k\ \Gamma$
**apply** (*induct n arbitrary*: *B* $\Gamma$)
**apply** (*case-tac B*)
**apply** (*case-tac* [*3*] *B*)
**apply** *simp-all*
**done**

**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*: $\Gamma$ $\Delta$) *simp-all*

**theorem** *decT-liftT* [*simp*]:
$k \leq k' \Longrightarrow k' + m \leq k + n \Longrightarrow \downarrow_\tau m\ k'\ (\uparrow_\tau n\ k\ \Gamma) = \uparrow_\tau (n - m)\ k\ \Gamma$
**apply** (*induct m arbitrary*: *n*)
**apply** (*subgoal-tac* [*2*] $k' + m \leq k + (n - Suc\ 0)$)
**apply** *simp-all*
**done**

**theorem** *decE-liftE* [*simp*]:
$k \leq k' \Longrightarrow k' + m \leq k + n \Longrightarrow \downarrow_e m\ k'\ (\uparrow_e n\ k\ \Gamma) = \uparrow_e (n - m)\ k\ \Gamma$
**apply** (*induct* $\Gamma$ *arbitrary*: $k\ k'$)
**apply** (*case-tac* [*2*] $a$)
**apply** *simp-all*
**done**

**theorem** *liftE0* [*simp*]: $\uparrow_e 0\ k\ \Gamma = \Gamma$
**apply** (*induct* $\Gamma$)
**apply** (*case-tac* [*2*] $a$)
**apply** *simp-all*
**done**

**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*: $k\ \Gamma$) *simp-all*

**lemma** *decE-length* [*simp*]: $\|\downarrow_e n\ k\ \Gamma\| = \|\Gamma\|$
**by** (*induct* $\Gamma$) *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*: $\Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Gamma \vdash_{wf} T \to U$
| *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*:
$\Gamma \vdash_{wf} TVar\ i$
$\Gamma \vdash_{wf} Top$
$\Gamma \vdash_{wf} T \to U$
$\Gamma \vdash_{wf} (\forall <: T.\ U)$

$\Gamma \vdash_{wf} (RcdT\ fTs)$

**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* $\Gamma' = map\ is\text{-}TVarB\ \Gamma \Longrightarrow$
$\Gamma\langle i\rangle = \lfloor TVarB\ T\rfloor \Longrightarrow \exists T.\ \Gamma'\langle i\rangle = \lfloor TVarB\ T\rfloor$
**apply** (*induct* $\Gamma$ *arbitrary*: $\Gamma'\ T\ i$)
**apply** *simp*
**apply** (*auto split*: *nat.split-asm*)
**apply** (*case-tac z*)
**apply** *simp-all*
**done**

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

**lemma** *wfE-replace*:
$\Delta$ @ $B :: \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf B} B' \Longrightarrow is\text{-}TVarB\ B' = is\text{-}TVarB\ B \Longrightarrow$
$\Delta$ @ $B' :: \Gamma \vdash_{wf}$
**apply** (*induct* $\Delta$)
**apply** *simp*
**apply** (*erule wf-Cons*)
**apply** (*erule well-formedE-cases*)
**apply** *assumption*
**apply** *simp*
**apply** (*erule well-formedE-cases*)
**apply** (*rule wf-Cons*)
**apply** (*case-tac a*)
**apply** *simp*
**apply** (*rule wf-equallength*)
**apply** *assumption*
**apply** *simp*
**apply** *simp*
**apply** (*rule wf-equallength*)
**apply** *assumption*
**apply** *simp*

**apply** *simp*
**done**

**lemma** *wf-weaken*:
**assumes** *H*: $\Delta$ @ $\Gamma \vdash_{wf} T$
**shows** $\uparrow_e$ (*Suc 0*) *0* $\Delta$ @ *B* :: $\Gamma \vdash_{wf} \uparrow_\tau$ (*Suc 0*) $\|\Delta\|$ *T*
**using** *H*
**apply** (*induct* $\Delta$ @ $\Gamma$ *T arbitrary*: $\Delta$)
**apply** *simp-all*
**apply** (*rule conjI*)
**apply** (*rule impI*)
**apply** (*rule wf-TVar*)
**apply** *simp*
**apply** (*rule impI*)
**apply** (*rule wf-TVar*)
**apply** (*subgoal-tac Suc i* − $\|\Delta\|$ = *Suc* (*i* − $\|\Delta\|$))
**apply** *simp*
**apply** *arith*
**apply** (*rule wf-Top*)
**apply** (*rule wf-arrow*)
**apply** *simp*
**apply** *simp*
**apply** (*rule wf-all*)
**apply** *simp*
**apply** *simp*
— records
**apply** (*rule wf-RcdT*)
**apply** *simp*
**apply** (*rule ballpI*)
**apply** (*drule liftrT-setD*)
**apply** (*erule exE conjE*)+
**apply** (*drule-tac x*=*l* **and** *y*=*T*[$\|\Delta\| \mapsto_\tau$ *Top*]$_\tau$ **in** *bpspec*)
**apply** *simp*+
**done**

**lemma** *wf-weaken'*: $\Gamma \vdash_{wf} T \Longrightarrow \Delta$ @ $\Gamma \vdash_{wf} \uparrow_\tau \|\Delta\|$ *0 T*
**apply** (*induct* $\Delta$)
**apply** *simp-all*
**apply** (*drule-tac B*=*a* **in** *wf-weaken* [*of* [], *simplified*])
**apply** *simp*
**done**

**lemma** *wfE-weaken*: $\Delta$ @ $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wfB} B \Longrightarrow \uparrow_e$ (*Suc 0*) *0* $\Delta$ @ *B* :: $\Gamma \vdash_{wf}$
**apply** (*induct* $\Delta$)
**apply** *simp*
**apply** (*rule wf-Cons*)
**apply** *assumption*+
**apply** *simp*
**apply** (*rule wf-Cons*)

**apply** (*erule well-formedE-cases*)
**apply** (*case-tac a*)
**apply** *simp*
**apply** (*rule wf-weaken*)
**apply** *assumption*
**apply** *simp*
**apply** (*rule wf-weaken*)
**apply** *assumption*
**apply** (*erule well-formedE-cases*)
**apply** *simp*
**done**

**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*
**apply** (*induct arbitrary*: *i*)
**apply** *simp*
**apply** (*simp split*: *nat.split-asm*)
**apply** (*frule-tac B=VarB T* **in** *wf-weaken* [*of* [], *simplified*])
**apply** *simp+*
**apply** (*rename-tac nat*)
**apply** (*drule-tac x=nat* **in** *meta-spec*)
**apply** *simp*
**apply** (*frule-tac T=*$\uparrow_\tau$ (*Suc nat*) *0 T* **in** *wf-weaken* [*of* [], *simplified*])
**apply** *simp*
**done**

**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* :: $\Gamma \vdash_{wf} snd\ (fT::fldT) \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow$
$\Delta[0 \mapsto_\tau U]_e$ @ $\Gamma \vdash_{wf} snd\ fT[\|\Delta\| \mapsto_\tau U]_\tau$
**apply** (*induct T* **and** *rT* **and** *fT arbitrary*: $\Delta$ **and** $\Delta$ **and** $\Delta$
*rule*: *liftT.induct liftrT.induct liftfT.induct*)
**apply** (*rename-tac nat* $\Delta$)
**apply** *simp-all*
**apply** (*rule conjI*)
**apply** (*rule impI*)
**apply** (*drule-tac* $\Gamma$=$\Gamma$ **and** $\Delta$=$\Delta[0 \mapsto_\tau U]_e$ **in** *wf-weaken'*)
**apply** *simp*
**apply** (*rule impI conjI*)+
**apply** (*erule well-formed-cases*)
**apply** (*rule wf-TVar*)
**apply** (*simp split*: *nat.split-asm*)
**apply** (*subgoal-tac* $\|\Delta\| \leq nat - Suc\ 0$)
**apply** (*rename-tac nata*)
**apply** (*subgoal-tac nat* $-$ *Suc* $\|\Delta\| = nata$)

**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *arith*
**apply** (*rule impI*)
**apply** (*erule well-formed-cases*)
**apply** (*rule wf-TVar*)
**apply** *simp*
**apply** (*rule wf-Top*)
**apply** (*erule well-formed-cases*)
**apply** (*rule wf-arrow*)
**apply** *simp+*
**apply** (*rename-tac type1 type2* $\Delta$)
**apply** (*erule well-formed-cases*)
**apply** (*rule wf-all*)
**apply** *simp*
**apply** (*thin-tac* $\bigwedge x$. *PROP P x* **for** *P* :: - $\Rightarrow$ *prop*)
**apply** (*drule-tac x*=*TVarB type1* :: $\Delta$ **in** *meta-spec*)
**apply** *simp*
**apply** (*erule well-formed-cases*)
**apply** (*rule wf-RcdT*)
**apply** *simp*
**apply** (*rule ballpI*)
**apply** (*drule substrT-setD*)
**apply** (*erule exE conjE*)+
**apply** (*drule meta-spec*)
**apply** (*drule meta-mp*)
**apply** *assumption*
**apply** (*thin-tac* $\forall x \in S$. *P x* **for** *S P*)
**apply** (*drule bpspec*)
**apply** *assumption*
**apply** *simp*
**apply** (*simp add*: *split-paired-all*)
**done**

**theorem** *wf-dec*: $\Delta$ @ $\Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} \downarrow_{\tau} \|\Delta\|\ 0\ T$
**apply** (*induct* $\Delta$ *arbitrary*: *T*)
**apply** *simp*
**apply** *simp*
**apply** (*drule wf-subst*(*1*) [*of* [], *simplified*])
**apply** (*rule wf-Top*)
**apply** *simp*
**done**

**theorem** *wfE-subst*: $\Delta$ @ *B* :: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Delta[0 \mapsto_{\tau} U]_e$ @ $\Gamma \vdash_{wf}$
**apply** (*induct* $\Delta$)
**apply** *simp*
**apply** (*erule well-formedE-cases*)
**apply** *assumption*
**apply** *simp*

**apply** (*case-tac a*)
**apply** (*erule well-formedE-cases*)
**apply** (*rule wf-Cons*)
**apply** *simp*
**apply** (*rule wf-subst*)
**apply** *assumption+*
**apply** *simp*
**apply** (*erule well-formedE-cases*)
**apply** (*rule wf-Cons*)
**apply** *simp*
**apply** (*rule wf-subst*)
**apply** *assumption+*
**done**

## 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*
**apply** (*rule H'*)
**apply** (*rule wf-subtype-env*)
**apply** (*rule H*)
**apply** (*rule wf-subtype* [*OF H*, *THEN conjunct1*])
**apply** (*rule wf-subtype* [*OF H*, *THEN conjunct2*])
**done**

**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 bexpI intro*!: *ballpI*
*elim*: *well-formed-cases ballpE elim*!: *bexpE*)+

**lemma** *subtype-weaken*:
**assumes** *H*: $\Delta\ @\ \Gamma \vdash P <: Q$
**and** *wf*: $\Gamma \vdash_{wf B} B$
**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 ↑e 1 0 Δ @ B :: Γ ⊢ TVar i <: ↑τ 1 ‖Δ‖ T
      by (rule subtyping.SA-trans-TVar)
    with True show ?thesis by simp
  next
    case False
    then have Suc i − ‖Δ‖ = Suc (i − ‖Δ‖) by arith
    with False SA-trans-TVar have (↑e 1 0 Δ @ B :: Γ)⟨Suc i⟩ = ⌊TVarB U⌋
      by simp
    moreover from False SA-trans-TVar
    have ↑e 1 0 Δ @ B :: Γ ⊢ ↑τ (Suc (Suc i)) 0 U <: ↑τ 1 ‖Δ‖ T
      by simp
    ultimately have ↑e 1 0 Δ @ B :: Γ ⊢ TVar (Suc i) <: ↑τ 1 ‖Δ‖ 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-all T1 S1 S2 T2)
  with SA-all(4) [of TVarB T1 :: Δ]
  show ?case by simp (iprover intro: subtyping.SA-all)
next
  case (SA-Rcd fs fs′)
  with wf have ↑e (Suc 0) 0 Δ @ B :: Γ ⊢wf by simp (rule wfE-weaken)
  moreover from ‹Δ @ Γ ⊢wf RcdT fs›
  have ↑e (Suc 0) 0 Δ @ B :: Γ ⊢wf ↑τ (Suc 0) ‖Δ‖ (RcdT fs)
    by (rule wf-weaken)
  hence ↑e (Suc 0) 0 Δ @ B :: Γ ⊢wf RcdT (↑rτ (Suc 0) ‖Δ‖ fs) by simp
  moreover from SA-Rcd have unique (↑rτ (Suc 0) ‖Δ‖ fs′) by simp
  moreover have ∀(l, T)∈set (↑rτ (Suc 0) ‖Δ‖ fs′).
    ∃S. (l, S)∈set (↑rτ (Suc 0) ‖Δ‖ fs) ∧ ↑e (Suc 0) 0 Δ @ B :: Γ ⊢ S <: T
  proof (rule ballpI)
    fix l T
    assume (l, T) ∈ set (↑rτ (Suc 0) ‖Δ‖ fs′)
    then obtain T′ where (l, T′) ∈ set fs′ and T: T = ↑τ (Suc 0) ‖Δ‖ T′
      by (blast dest: liftrT-setD)
    with SA-Rcd obtain S where
      lS: (l, S) ∈ set fs
       and ST: ↑e (Suc 0) 0 Δ @ B :: Γ ⊢ ↑τ (Suc 0) ‖Δ‖ S <: ↑τ (Suc 0) ‖Δ‖
(T[‖Δ‖ ↦τ Top]τ)
      by fastforce
    with T have ↑e (Suc 0) 0 Δ @ B :: Γ ⊢ ↑τ (Suc 0) ‖Δ‖ S <: ↑τ (Suc 0) ‖Δ‖
T′
      by simp
    moreover from lS have (l, ↑τ (Suc 0) ‖Δ‖ S) ∈ set (↑rτ (Suc 0) ‖Δ‖ fs)
      by (rule liftrT-set)
    moreover note T
    ultimately show ∃S. (l, S)∈set (↑rτ (Suc 0) ‖Δ‖ fs) ∧ ↑e (Suc 0) 0 Δ @ B
```

```
:: Γ ⊢ S <: T
      by auto
  qed
  ultimately have ↑e (Suc 0) 0 Δ @ B :: Γ ⊢ RcdT (↑rτ (Suc 0) ‖Δ‖ fs) <: RcdT
(↑rτ (Suc 0) ‖Δ‖ fs')
    by (rule subtyping.SA-Rcd)
  thus ?case by simp
qed

lemma subtype-weaken': — A.2
  Γ ⊢ P <: Q ⟹ Δ @ Γ ⊢wf ⟹ Δ @ Γ ⊢ ↑τ ‖Δ‖ 0 P <: ↑τ ‖Δ‖ 0 Q
  apply (induct Δ)
  apply simp-all
  apply (erule well-formedE-cases)
  apply simp
  apply (drule-tac B=a and Γ=Δ @ Γ in subtype-weaken [of [], simplified])
  apply simp-all
  done

lemma fieldT-size [simp]:
  (a, T) ∈ set fs ⟹ size T < Suc (size-list (size-prod (λx. 0) size) fs)
  by (induct fs arbitrary: a T rule: list.induct) fastforce+

lemma subtype-trans: — A.3
  Γ ⊢ S <: Q ⟹ Γ ⊢ Q <: T ⟹ Γ ⊢ S <: T
  Δ @ TVarB Q :: Γ ⊢ M <: N ⟹ Γ ⊢ P <: Q ⟹
     Δ @ TVarB P :: Γ ⊢ M <: N
  using wf-measure-size
proof (induct Q arbitrary: Γ S T Δ P M N rule: wf-induct-rule)
  case (less Q)
  {
    fix Γ S T Q'
    assume Γ ⊢ S <: Q'
    then have Γ ⊢ Q' <: T ⟹ size Q = size Q' ⟹ Γ ⊢ 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 Γ T1 S1 S2 T2)
      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 T₁' T₂')
    from SA-arrow SA-arrow' have Γ ⊢ S₁ → S₂ <: T₁' → T₂'
      by (auto intro!: subtyping.SA-arrow intro: less(1) [of T₁] less(1) [of T₂])
    with SA-arrow show ?thesis by simp
  qed
next
  case (SA-all Γ T₁ S₁ S₂ T₂)
  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 T₁' T₂')
    from SA-all SA-all' have Γ ⊢ T₁' <: S₁
      by − (rule less(1), simp-all)
    moreover from SA-all SA-all' have TVarB T₁' :: Γ ⊢ S₂ <: T₂
      by − (rule less(2) [of - []], simplified], simp-all)
    with SA-all SA-all' have TVarB T₁' :: Γ ⊢ S₂ <: T₂'
      by − (rule less(1), simp-all)
    ultimately have Γ ⊢ (∀ <:S₁. S₂) <: (∀ <:T₁'. T₂')
      by (rule subtyping.SA-all)
    with SA-all show ?thesis by simp
  qed
next
  case (SA-Rcd Γ fs₁ fs₂)
  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 fs₂')
    note ‹Γ ⊢wf›
    moreover note ‹Γ ⊢wf RcdT fs₁›
    moreover note ‹unique fs₂'›
    moreover have ∀ (l, T)∈set fs₂'. ∃ S. (l, S)∈set fs₁ ∧ Γ ⊢ S <: T
    proof (rule ballpI)
      fix l T
      assume lT: (l, T) ∈ set fs₂'
      with SA-Rcd obtain U where
        fs2: (l, U) ∈ set fs₂ and UT: Γ ⊢ U <: T by blast
      with SA-Rcd SA-Rcd' obtain S where
        fs1: (l, S) ∈ set fs₁ and SU: Γ ⊢ S <: U by blast
      from SA-Rcd SA-Rcd' fs2 have (U, Q) ∈ measure size by simp
```

```
        hence Γ ⊢ S <: T using SU UT by (rule less(1))
        with fs1 show ∃S. (l, S)∈set fs₁ ∧ Γ ⊢ S <: T by blast
       qed
       ultimately have Γ ⊢ RcdT fs₁ <: RcdT fs₂' 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 :: Γ ⊢ 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 < ‖Δ‖)
     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 = ‖Δ‖)
      case True
      from SA-trans-TVar have (Δ @ [TVarB P]) @ Γ ⊢_wf
       by (auto intro: wfE-replace elim!: wf-subtypeE)
      with ‹Γ ⊢ P <: Q›
       have (Δ @ [TVarB P]) @ Γ ⊢ ↑_τ ‖Δ @ [TVarB P]‖ 0 P <: ↑_τ ‖Δ @
[TVarB P]‖ 0 Q
        by (rule subtype-weaken')
      with SA-trans-TVar True False have Δ @ TVarB P :: Γ ⊢ ↑_τ (Suc ‖Δ‖)
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
 apply (induct Δ @ TVarB Q :: Γ S T arbitrary: Δ)
 apply simp-all
 apply (rule SA-Top)
 apply (rule wfE-subst)
 apply assumption
 apply (erule wf-subtypeE)
 apply assumption
 apply (rule wf-subst)
 apply assumption
 apply (erule wf-subtypeE)
 apply assumption
 apply (rule impI conjI)+
 apply (rule subtype-refl)
 apply (rule wfE-subst)
 apply assumption
 apply (erule wf-subtypeE)
```

**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** (*drule-tac* $T{=}P$ **and** $\Delta{=}\Delta[0 \mapsto_\tau P]_e$ **in** *wf-weaken′*)
**apply** *simp*
**apply** (*rule conjI impI*)+
**apply** (*rule SA-refl-TVar*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** (*drule wf-subst*)
**apply** *assumption*
**apply** *simp*
**apply** (*rule impI*)
**apply** (*rule SA-refl-TVar*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** (*drule wf-subst*)
**apply** *assumption*
**apply** *simp*
**apply** (*rule conjI impI*)+
**apply** *simp*
**apply** (*drule-tac* $\Gamma{=}\Gamma$ **and** $\Delta{=}\Delta[0 \mapsto_\tau P]_e$ **in** *subtype-weaken′*)
**apply** (*erule wf-subtypeE*)+
**apply** *assumption*
**apply** *simp*
**apply** (*rule subtype-trans*(*1*))
**apply** *assumption*+
**apply** (*rule conjI impI*)+
**apply** (*rule SA-trans-TVar*)
**apply** (*simp split*: *nat.split-asm*)
**apply** (*subgoal-tac* $\|\Delta\| \leq i - Suc\ 0$)
**apply** (*rename-tac nat*)
**apply** (*subgoal-tac* $i - Suc\ \|\Delta\| = nat$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *arith*
**apply** *simp*
**apply** (*rule impI*)
**apply** (*rule SA-trans-TVar*)
**apply** (*simp split*: *nat.split-asm*)
**apply** (*subgoal-tac Suc* $(\|\Delta\| - Suc\ 0) = \|\Delta\|$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** (*rule SA-arrow*)

**apply** *simp+*
**apply** (*rule SA-all*)
**apply** *simp*
**apply** *simp*
**apply** (*erule wf-subtypeE*)
**apply** (*rule SA-Rcd*)
**apply** (*erule wfE-subst*)
**apply** *assumption*
**apply** (*drule wf-subst*)
**apply** *assumption*
**apply** *simp*
**apply** *simp*
**apply** (*rule ballpI*)
**apply** (*drule substrT-setD*)
**apply** (*erule exE conjE*)+
**apply** (*drule bpspec*)
**apply** *assumption*
**apply** *simp*
**apply** (*erule exE*)
**apply** (*erule conjE*)+
**apply** (*rule exI*)
**apply** (*rule conjI*)
**apply** (*erule substrT-set*)
**apply** *assumption*
**done**

**lemma** *subst-subtype*:
**assumes** *H*: $\Delta$ @ *VarB V* :: $\Gamma \vdash T <: U$
**shows** $\downarrow_e$ *1 0* $\Delta$ @ $\Gamma \vdash \downarrow_\tau$ *1* $\|\Delta\|$ *T* $<: \downarrow_\tau$ *1* $\|\Delta\|$ *U*
**using** *H*
**apply** (*induct* $\Delta$ @ *VarB V* :: $\Gamma$ *T U arbitrary*: $\Delta$)
**apply** *simp-all*
**apply** (*rule SA-Top*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)
**apply** (*rule wf-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)
**apply** (*rule impI conjI*)+
**apply** (*rule SA-Top*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)+
**apply** (*rule conjI impI*)+
**apply** (*rule SA-refl-TVar*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)

**apply** (*drule wf-subst*)
**apply** (*rule wf-Top*)
**apply** *simp*
**apply** (*rule impI*)
**apply** (*rule SA-refl-TVar*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)
**apply** (*drule wf-subst*)
**apply** (*rule wf-Top*)
**apply** *simp*
**apply** (*rule conjI impI*)+
**apply** *simp*
**apply** (*rule conjI impI*)+
**apply** (*simp split*: *nat.split-asm*)
**apply** (*rule SA-trans-TVar*)
**apply** (*subgoal-tac* $\|\Delta\| \leq i - Suc\ 0$)
**apply** (*rename-tac nat*)
**apply** (*subgoal-tac* $i - Suc\ \|\Delta\| = nat$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *arith*
**apply** *simp*
**apply** (*rule impI*)
**apply** (*rule SA-trans-TVar*)
**apply** *simp*
**apply** (*subgoal-tac* $0 < \|\Delta\|$)
**apply** *simp*
**apply** *arith*
**apply** (*rule SA-arrow*)
**apply** *simp*+
**apply** (*rule SA-all*)
**apply** *simp*
**apply** *simp*
**apply** (*rule SA-Rcd*)
**apply** (*erule wfE-subst*)
**apply** (*rule wf-Top*)
**apply** (*drule wf-subst*)
**apply** (*rule wf-Top*)
**apply** *simp*
**apply** *simp*
**apply** (*rule ballpI*)
**apply** (*drule substrT-setD*)
**apply** (*erule exE conjE*)+
**apply** (*drule bpspec*)
**apply** *assumption*
**apply** *simp*
**apply** (*erule exE*)
**apply** (*erule conjE*)+

**apply** (*rule exI*)
**apply** (*rule conjI*)
**apply** (*erule substrT-set*)
**apply** *assumption*
**done**

## 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 $\Delta$, 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 $\Delta$ 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*: ⊢ *PVar T* : *T* ⇒ [*VarB T*]
| *P-Rcd*: ⊢ *fps* [:] *fTs* ⇒ Δ ⟹ ⊢ *PRcd fps* : *RcdT fTs* ⇒ Δ
| *P-Nil*: ⊢ [] [:] [] ⇒ []
| *P-Cons*: ⊢ *p* : *T* ⇒ Δ$_1$ ⟹ ⊢ *fps* [:] *fTs* ⇒ Δ$_2$ ⟹ *fps*⟨*l*⟩$_?$ = ⊥ ⟹
⊢ ((*l*, *p*) :: *fps*) [:] ((*l*, *T*) :: *fTs*) ⇒ ↑$_e$ ‖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*: Γ ⊢$_{wf}$ ⟹ Γ⟨*i*⟩ = ⌊*VarB U*⌋ ⟹ *T* = ↑$_\tau$ (*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*$_1$ → ↓$_\tau$ *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*$_1$ ∘$_\tau$ *T*$_2$ : *T*$_{12}$[*0* ↦$_\tau$ *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*$_2$) : ↓$_\tau$ ‖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*: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash [] \; [:] \; []$
| *T-Cons*: $\Gamma \vdash t : T \Longrightarrow \Gamma \vdash fs \; [:] \; fTs \Longrightarrow fs\langle l\rangle_? = \bot \Longrightarrow$
$\Gamma \vdash (l,\ t) :: fs \; [:] \; (l,\ T) :: fTs$

**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*:
$\Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wf} T$
$\Gamma' \vdash fs \; [:] \; fTs \Longrightarrow (\forall (l,\ T) \in set\ fTs.\ \Gamma' \vdash_{wf} T) \wedge$
$unique\ fTs \wedge (\forall l.\ (fs\langle l\rangle_? = \bot) = (fTs\langle l\rangle_? = \bot))$
**apply** (*induct set*: *typing typings*)
**apply** *simp*
**apply** (*rule wf-liftB*)
**apply** *assumption*+
**apply** (*drule wf-typeE1*)+
**apply** (*erule well-formedE-cases*)+
**apply** (*rule wf-arrow*)
**apply** *simp*
**apply** *simp*
**apply** (*rule wf-subst* [*of* [], *simplified*])
**apply** *assumption*
**apply** (*rule wf-Top*)
**apply** (*erule well-formed-cases*)
**apply** *assumption*
**apply** (*rule wf-all*)
**apply** (*drule wf-typeE1*)
**apply** (*erule well-formedE-cases*)
**apply** *simp*
**apply** *assumption*
**apply** (*erule well-formed-cases*)
**apply** (*rule wf-subst* [*of* [], *simplified*])
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
— records
**apply** (*erule wf-dec*)
**apply** (*erule conjE*)+
**apply** (*rule wf-RcdT*)
**apply** *assumption*+
**apply** (*erule well-formed-cases*)
**apply** (*blast dest*: *assoc-set*)
**apply** *simp*
**apply** *simp*
**done**

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

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

**lemmas** *typing-induct* = *typing-typings.inducts*(*1*)
[*of* - - - - λ*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* - - - λ*x y z*. *True*, *simplified True-simps*, *consumes 1*,
*case-names T-Nil T-Cons*]

**lemma** *narrow-type*: — A.7
Δ @ *TVarB Q* :: Γ ⊢ *t* : *T* ⟹
Γ ⊢ *P* <: *Q* ⟹ Δ @ *TVarB P* :: Γ ⊢ *t* : *T*
Δ @ *TVarB Q* :: Γ ⊢ *ts* [:] *Ts* ⟹
Γ ⊢ *P* <: *Q* ⟹ Δ @ *TVarB P* :: Γ ⊢ *ts* [:] *Ts*
**apply** (*induct* Δ @ *TVarB Q* :: Γ *t T* **and** Δ @ *TVarB Q* :: Γ *ts Ts*
*arbitrary*: Δ **and** Δ *set*: *typing typings*)
**apply** *simp-all*
**apply** (*rule T-Var*)
**apply** (*erule wfE-replace*)
**apply** (*erule wf-subtypeE*)
**apply** *simp*+
**apply** (*case-tac i* < ‖Δ‖)
**apply** *simp*
**apply** (*case-tac i* = ‖Δ‖)
**apply** *simp*
**apply** (*simp split*: *nat.split nat.split-asm*)+
**apply** (*rule T-Abs* [*simplified*])
**apply** *simp*
**apply** (*rule-tac* $T_{11}$=$T_{11}$ **in** *T-App*)
**apply** *simp*+
**apply** (*rule T-TAbs*)
**apply** *simp*
**apply** (*rule-tac* $T_{11}$=$T_{11}$ **in** *T-TApp*)
**apply** *simp*
**apply** (*rule subtype-trans*(*2*))
**apply** *assumption*+
**apply** (*rule-tac S*=*S* **in** *T-Sub*)
**apply** *simp*
**apply** (*rule subtype-trans*(*2*))
**apply** *assumption*+
— records

**apply** (*rule T-Let*)
**apply** *blast*
**apply** *assumption*
**apply** *simp*
**apply** (*rule T-Rcd*)
**apply** *simp*
**apply** (*rule T-Proj*)
**apply** *blast*
**apply** *assumption*
**apply** (*rule T-Nil*)
**apply** (*erule wfE-replace*)
**apply** (*erule wf-subtypeE*)
**apply** *simp+*
**apply** (*rule T-Cons*)
**apply** *simp+*
**done**

**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** $\Gamma \vdash T <: T$
**proof** (*rule subtype-refl*)
**from** *t* **show** $\Gamma \vdash_{wf}$ **by** (*rule wf-typeE1*)
**from** *t* **show** $\Gamma \vdash_{wf} T$ **by** (*rule wf-typeE2*)
**qed**

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

**by** (*rule T-Sub*) (*rule T-Sub*(*5*))
**qed**

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

**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* $\Gamma$ $\lambda{<:}S.\ s\ T$ *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*])
**with** *ty1* **show** *?case* **using** *ty2* **by** (*rule T-TAbs*)
**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*))
**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*: $\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow \mathit{TVarB}\ U :: \Gamma \vdash s : S' \Longrightarrow$
$\mathit{TVarB}\ U :: \Gamma \vdash S' <: U' \Longrightarrow P$
**shows** *P* **using** *H subtype-refl'* [*OF H*]
**by** (*rule TAbs-type*) (*rule R*)

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** *H*: $\Gamma \vdash t : T$
**shows** $t = \mathit{Rcd}\ fs \Longrightarrow \Gamma \vdash T <: \mathit{RcdT}\ fTs \Longrightarrow$
$\forall (l,\ U) \in \mathit{set}\ fTs.\ \exists u.\ fs\langle l\rangle_? = \lfloor u \rfloor \wedge \Gamma \vdash u : U$
**using** *H*
**apply** (*induct arbitrary*: *fs fTs rule*: *typing-induct*, *simp-all*)
**apply** (*drule meta-spec*)+

**apply** (*drule meta-mp*)
**apply** (*rule refl*)
**apply** (*erule meta-mp*)
**apply** (*rule subtype-trans*(*1*))
**apply** *assumption*+
**apply** (*erule subtyping.cases*)
**apply** *simp-all*
**apply** (*rule ballpI*)
**apply** (*drule bpspec*)
**apply** *assumption*
**apply** (*erule exE conjE*)+
**apply** (*drule typings-setD*)
**apply** *assumption*
**apply** (*erule exE conjE*)+
**apply** (*rule exI conjI*)+
**apply** *simp*
**apply** (*erule T-Sub*)
**apply** *assumption*
**done**

**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,

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

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*:
$\Gamma \vdash$ *Rcd fs* : $T \Longrightarrow \Gamma \vdash T <:$ *RcdT fTs* $\Longrightarrow$
$\Gamma \vdash$ *map* $(\lambda(l,\ T).\ (l,$ *the* (*fs*$\langle l\rangle_?$))) *fTs* [:] *fTs*
**apply** (*drule Rcd-type1*)
**apply** (*rule refl*)
**apply** *assumption*
**apply** (*induct fTs rule*: *list.induct*)
**apply** *simp*

**apply** (*rule T-Nil*)
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*simp add*: *split-paired-all*)
**apply** (*rule T-Cons*)
**apply** (*drule-tac x*=*a* **and** *y*=*b* **in** *bpspec*)
**apply** *simp*
**apply** (*erule exE conjE*)+
**apply** *simp*
**apply** (*rename-tac list*)
**apply** (*subgoal-tac* $\Gamma \vdash$ *RcdT* ((*a*, *b*) :: *list*) <: *RcdT list*)
**apply** (*erule meta-mp*)
**apply** (*erule subtype-trans*(*1*))
**apply** *assumption*
**apply** (*erule wf-subtypeE*)
**apply** (*rule SA-Rcd*)
**apply** *assumption*+
**apply** (*erule well-formed-cases*)
**apply** *simp*
**apply** (*rule ballpI*)
**apply** (*rule exI*)
**apply** (*rule conjI*)
**apply** (*rule-tac* [*2*] *subtype-refl*)
**apply** *simp*
**apply** *assumption*
**apply** (*erule well-formed-cases*)
**apply** (*erule-tac x*=*l* **and** *y*=*Ta* **in** *bpspec*)
**apply** *simp*
**apply** (*erule wf-subtypeE*)
**apply** (*erule well-formed-cases*)
**apply** *simp*
**done**

**lemma** *Rcd-type2'*:
**assumes** *H*: $\Gamma \vdash$ *Rcd fs* : *RcdT fTs*
**shows** $\Gamma \vdash$ *map* ($\lambda$(*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*:
$\vdash p : T \Rightarrow \Delta \Longrightarrow \vdash \uparrow_p n\ k\ p : \uparrow_\tau n\ k\ T \Rightarrow \uparrow_e n\ k\ \Delta$
$\vdash fps$ [:] $fTs \Rightarrow \Delta \Longrightarrow \vdash \uparrow_{rp} n\ k\ fps$ [:] $\uparrow_{r\tau} n\ k\ fTs \Rightarrow \uparrow_e n\ k\ \Delta$

**apply** (*induct set*: *ptyping ptypings*)
**apply** *simp-all*
**apply** (*rule P-Var*)
**apply** (*erule P-Rcd*)
**apply** (*rule P-Nil*)
**apply** (*drule-tac* $p{=}\uparrow_p\ n\ k\ p$ **and** $fps{=}\uparrow_{rp}\ n\ k\ fps$ **in** *P-Cons*)
**apply** *simp-all*
**done**

**lemma** *type-weaken*:
$\Delta\ @\ \Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wfB} B \Longrightarrow$
$\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma \vdash \uparrow\ 1\ \|\Delta\|\ t : \uparrow_\tau\ 1\ \|\Delta\|\ T$
$\Delta\ @\ \Gamma \vdash fs\ [:]\ fTs \Longrightarrow \Gamma \vdash_{wfB} B \Longrightarrow$
$\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma \vdash \uparrow_r\ 1\ \|\Delta\|\ fs\ [:]\ \uparrow_{r\tau}\ 1\ \|\Delta\|\ fTs$
**apply** (*induct* $\Delta\ @\ \Gamma\ t\ T$ **and** $\Delta\ @\ \Gamma\ fs\ fTs$
*arbitrary*: $\Delta$ **and** $\Delta$ *set*: *typing typings*)
**apply** *simp-all*
**apply** (*rule conjI*)
**apply** (*rule impI*)
**apply** (*rule T-Var*)
**apply** (*erule wfE-weaken*)
**apply** *simp+*
**apply** (*rule impI*)
**apply** (*rule T-Var*)
**apply** (*erule wfE-weaken*)
**apply** *assumption*
**apply** (*subgoal-tac Suc* $i - \|\Delta\| = Suc\ (i - \|\Delta\|)$)
**apply** *simp*
**apply** *arith*
**apply** (*rule refl*)
**apply** (*rule T-Abs* [*simplified*])
**apply** *simp*
**apply** (*rule-tac* $T_{11}{=}\uparrow_\tau\ (Suc\ 0)\ \|\Delta\|\ T_{11}$ **in** *T-App*)
**apply** *simp*
**apply** *simp*
**apply** (*rule T-TAbs*)
**apply** *simp*
**apply** (*erule-tac T-TApp* [*THEN T-eq*])
**apply** (*drule subtype-weaken*)
**apply** *simp+*
**apply** (*case-tac* $\Delta$)
**apply** (*simp add*: *liftT-substT-strange* [*of - 0*, *simplified*])+
**apply** (*rule-tac* $S{=}\uparrow_\tau\ (Suc\ 0)\ \|\Delta\|\ S$ **in** *T-Sub*)
**apply** *simp*
**apply** (*drule subtype-weaken*)
**apply** *simp+*
— records
**apply** (*drule-tac* $\Gamma{=}\uparrow_e\ (Suc\ 0)\ 0\ \Delta'\ @\ B :: \Gamma$ **in** *T-Let*)
**apply** (*erule lift-ptyping*)

**apply** *assumption*
**apply** (*simp add*: *ac-simps*)
**apply** (*rule T-Rcd*)
**apply** *simp*
**apply** (*rule-tac fTs*=$\uparrow_{r\tau}$ (*Suc 0*) $\|\Delta\|$ *fTs* **in** *T-Proj*)
**apply** *simp*
**apply** (*erule-tac liftrT-assoc-Some*)
**apply** (*rule T-Nil*)
**apply** (*erule wfE-weaken*)
**apply** *assumption*
**apply** (*rule T-Cons*)
**apply** *simp+*
**done**

**lemma** *type-weaken'*: — A.5(6)
$\Gamma \vdash t : T \Longrightarrow \Delta \mathbin{@} \Gamma \vdash_{wf} \Longrightarrow \Delta \mathbin{@} \Gamma \vdash \uparrow \|\Delta\|\ 0\ t : \uparrow_\tau \|\Delta\|\ 0\ T$
**apply** (*induct* $\Delta$)
**apply** *simp*
**apply** *simp*
**apply** (*erule well-formedE-cases*)
**apply** *simp*
**apply** (*drule-tac B*=*a* **in** *type-weaken*(*1*) [*of* [], *simplified*])
**apply** *simp+*
**done**

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*:
$\vdash p : T \Rightarrow \Delta \Longrightarrow \vdash p[k \mapsto_\tau U]_p : T[k \mapsto_\tau U]_\tau \Rightarrow \Delta[k \mapsto_\tau U]_e$
$\vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow \vdash fps[k \mapsto_\tau U]_{rp}\ [:]\ fTs[k \mapsto_\tau U]_{r\tau} \Rightarrow \Delta[k \mapsto_\tau U]_e$
**apply** (*induct set*: *ptyping ptypings*)
**apply** *simp-all*
**apply** (*rule P-Var*)
**apply** (*erule P-Rcd*)
**apply** (*rule P-Nil*)
**apply** (*drule-tac p*=$p[k \mapsto_\tau U]_p$ **and** *fps*=$fps[k \mapsto_\tau U]_{rp}$ **in** *P-Cons*)
**apply** *simp+*
**done**

**theorem** *subst-type*: — A.8
$\Delta \mathbin{@} VarB\ U :: \Gamma \vdash t : T \Longrightarrow \Gamma \vdash u : U \Longrightarrow$
$\downarrow_e 1\ 0\ \Delta \mathbin{@} \Gamma \vdash t[\|\Delta\| \mapsto u] : \downarrow_\tau 1\ \|\Delta\|\ T$
$\Delta \mathbin{@} VarB\ U :: \Gamma \vdash fs\ [:]\ fTs \Longrightarrow \Gamma \vdash u : U \Longrightarrow$
$\downarrow_e 1\ 0\ \Delta \mathbin{@} \Gamma \vdash fs[\|\Delta\| \mapsto u]_r\ [:]\ \downarrow_{r\tau} 1\ \|\Delta\|\ fTs$
**apply** (*induct* $\Delta \mathbin{@} VarB\ U :: \Gamma\ t\ T$ **and** $\Delta \mathbin{@} VarB\ U :: \Gamma\ fs\ fTs$
*arbitrary*: $\Delta$ **and** $\Delta$ *set*: *typing typings*)
**apply** *simp*
**apply** (*rule conjI*)
**apply** (*rule impI*)

**apply** *simp*
**apply** (*drule-tac* $\Delta{=}\Delta[0 \mapsto_\tau Top]_e$ **in** *type-weaken'*)
**apply** (*rule wfE-subst*)
**apply** *assumption*
**apply** (*rule wf-Top*)
**apply** *simp*
**apply** (*rule impI conjI*)+
**apply** (*simp split*: *nat.split-asm*)
**apply** (*rule T-Var*)
**apply** (*erule wfE-subst*)
**apply** (*rule wf-Top*)
**apply** (*subgoal-tac* $\|\Delta\| \leq i - Suc\ 0$)
**apply** (*rename-tac nat*)
**apply** (*subgoal-tac* $i - Suc\ \|\Delta\| = nat$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *arith*
**apply** *simp*
**apply** (*rule impI*)
**apply** (*rule T-Var*)
**apply** (*erule wfE-subst*)
**apply** (*rule wf-Top*)
**apply** *simp*
**apply** (*subgoal-tac Suc* $(\|\Delta\| - Suc\ 0) = \|\Delta\|$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *simp*
**apply** (*rule T-Abs* [*THEN T-eq*])
**apply** *simp*
**apply** (*simp add*: *substT-substT* [*symmetric*])
**apply** *simp*
**apply** (*rule-tac* $T_{11}{=}T_{11}[\|\Delta\| \mapsto_\tau Top]_\tau$ **in** *T-App*)
**apply** *simp*+
**apply** (*rule T-TAbs*)
**apply** *simp*
**apply** *simp*
**apply** (*rule T-TApp* [*THEN T-eq*])
**apply** *simp*
**apply** (*rule subst-subtype* [*simplified*])
**apply** *assumption*
**apply** (*simp add*: *substT-substT* [*symmetric*])
**apply** (*rule-tac* $S{=}S[\|\Delta\| \mapsto_\tau Top]_\tau$ **in** *T-Sub*)
**apply** *simp*
**apply** *simp*
**apply** (*rule subst-subtype* [*simplified*])
**apply** *assumption*
— records
**apply** *simp*
**apply** (*drule-tac* $\Gamma{=}\Delta'[0 \mapsto_\tau Top]_e$ @ $\Gamma$ **in** *T-Let*)

**apply** (*erule subst-ptyping*)
**apply** *simp*
**apply** (*simp add*: *ac-simps*)
**apply** *simp*
**apply** (*rule T-Rcd*)
**apply** *simp*
**apply** *simp*
**apply** (*rule-tac fTs=fTs*[$\|\Delta\| \mapsto_\tau$ *Top*]$_{r\tau}$ **in** *T-Proj*)
**apply** *simp*
**apply** (*erule-tac substrTT-assoc-Some*)
**apply** *simp*
**apply** (*rule T-Nil*)
**apply** (*erule wfE-subst*)
**apply** (*rule wf-Top*)
**apply** *simp*
**apply** (*rule T-Cons*)
**apply** *simp+*
**done**

**theorem** *substT-type*: — A.11
$\Delta$ @ *TVarB Q* :: $\Gamma \vdash t : T \Longrightarrow \Gamma \vdash P <: Q \Longrightarrow$
$\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash t[\|\Delta\| \mapsto_\tau P] : T[\|\Delta\| \mapsto_\tau P]_\tau$
$\Delta$ @ *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*[$\|\Delta\| \mapsto_\tau P$]$_{r\tau}$
**apply** (*induct* $\Delta$ @ *TVarB Q* :: $\Gamma$ *t T* **and** $\Delta$ @ *TVarB Q* :: $\Gamma$ *fs fTs*
*arbitrary*: $\Delta$ **and** $\Delta$ *set*: *typing typings*)
**apply** *simp-all*
**apply** (*rule impI conjI*)+
**apply** (*rule T-Var*)
**apply** (*erule wfE-subst*)
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*simp split*: *nat.split-asm*)
**apply** (*subgoal-tac* $\|\Delta\| \leq i - Suc\ 0$)
**apply** (*rename-tac nat*)
**apply** (*subgoal-tac* $i - Suc\ \|\Delta\| = nat$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *arith*
**apply** *simp*
**apply** (*rule impI*)
**apply** (*case-tac* $i = \|\Delta\|$)
**apply** *simp*
**apply** (*rule T-Var*)
**apply** (*erule wfE-subst*)
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** *simp*
**apply** (*subgoal-tac* $i < \|\Delta\|$)

**apply** (*subgoal-tac Suc* ($\|\Delta\|$ − *Suc 0*) = $\|\Delta\|$)
**apply** (*simp* (*no-asm-simp*))
**apply** *arith*
**apply** *arith*
**apply** (*rule T-Abs* [*THEN T-eq*])
**apply** *simp*
**apply** (*simp add*: *substT-substT* [*symmetric*])
**apply** (*rule-tac* $T_{11}$=$T_{11}[\|\Delta\| \mapsto_\tau P]_\tau$ **in** *T-App*)
**apply** *simp+*
**apply** (*rule T-TAbs*)
**apply** *simp*
**apply** (*rule T-TApp* [*THEN T-eq*])
**apply** *simp*
**apply** (*rule substT-subtype*)
**apply** *assumption*
**apply** *assumption*
**apply** (*simp add*: *substT-substT* [*symmetric*])
**apply** (*rule-tac S*=$S[\|\Delta\| \mapsto_\tau P]_\tau$ **in** *T-Sub*)
**apply** *simp*
**apply** (*rule substT-subtype*)
**apply** *assumption*
**apply** *assumption*
— records
**apply** (*drule-tac* $\Gamma$=$\Delta'[0 \mapsto_\tau P]_e$ @ $\Gamma$ **in** *T-Let*)
**apply** (*erule subst-ptyping*)
**apply** *simp*
**apply** (*simp add*: *ac-simps*)
**apply** (*rule T-Rcd*)
**apply** *simp*
**apply** (*rule-tac fTs*=$fTs[\|\Delta\| \mapsto_\tau P]_{r\tau}$ **in** *T-Proj*)
**apply** *simp*
**apply** (*erule-tac substrTT-assoc-Some*)
**apply** (*rule T-Nil*)
**apply** (*erule wfE-subst*)
**apply** (*erule wf-subtypeE*)
**apply** *assumption*
**apply** (*rule T-Cons*)
**apply** *simp+*
**done**

## 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*: (λ:*T*. *t*) ∈ *value*
| *TAbs*: (λ<:*T*. *t*) ∈ *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 $\vdash fps\ [\rhd]\ fs \Rightarrow ts$ for matching a list of field patterns *fps* against a list of fields *fs*:

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

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

**inductive**
  *eval* :: $trm \Rightarrow trm \Rightarrow bool$ (**infixl** $\longmapsto$ *50*)
  **and** *evals* :: $rcd \Rightarrow rcd \Rightarrow bool$ (**infixl** $[\longmapsto]$ *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*: $v \in value \Longrightarrow \vdash p \rhd v \Rightarrow ts \Longrightarrow (LET\ p = v\ IN\ t) \longmapsto t[0 \mapsto_s ts]$
| *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*
**apply** (*induct rule*: *matchs-induct*)
**apply** (*rule M-Nil*)
**apply** (*simp split*: *if-split-asm*)
**apply** (*rule M-Cons*)
**apply** *simp-all*
**done**

**lemma** *matchs-eq*:
**assumes** $H$: $\vdash$ *fps* [$\triangleright$] *fs* $\Rightarrow$ *ts*
**shows** $\forall$ (*l*, *p*) $\in$ *set fps*. *fs*$\langle l\rangle_?$ = *fs*′$\langle l\rangle_?$ $\Longrightarrow$ $\vdash$ *fps* [$\triangleright$] *fs*′ $\Rightarrow$ *ts*
**using** *H*
**apply** (*induct rule*: *matchs-induct*)
**apply** (*rule M-Nil*)
**apply** (*rule M-Cons*)
**apply** *auto*
**done**

**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* [$\triangleright$] *fs* $\Rightarrow$ *ts* $\Longrightarrow$ $\vdash$ *fps* [$\triangleright$] *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* [$\triangleright$] *map* ($\lambda$(*l*, *T*). (*l*, *the* (*fs*$\langle l\rangle_?$))) *fTs* $\Rightarrow$ *ts* $\Longrightarrow$ $\vdash$ *fps* [$\triangleright$] *fs* $\Rightarrow$ *ts*
**by** (*rule matchs-eq* [*OF* - *reorder-eq* [*THEN ball-eq-sym*]], *assumption*+)

**theorem** *matchs-tl*:
**assumes** $H$: $\vdash$ *fps* [$\triangleright$] (*l*, *t*) :: *fs* $\Rightarrow$ *ts*
**shows** *fps*$\langle l\rangle_?$ = $\bot$ $\Longrightarrow$ $\vdash$ *fps* [$\triangleright$] *fs* $\Rightarrow$ *ts*
**using** *H*
**apply** (*induct fps* (*l*, *t*) :: *fs ts arbitrary*: *l t fs rule*: *matchs-induct*)
**apply** (*simp-all split*: *if-split-asm*)
**apply** (*rule M-Nil*)
**apply** (*rule M-Cons*)
**apply** *auto*
**done**

**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** $fTs\ \Delta$ *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* [$\triangleright$] (*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** $(\Gamma_1 \mathbin{@} \uparrow_e \|\Delta_1\|\ 0\ \Delta_2) \mathbin{@} \Delta_1 \mathbin{@} \Gamma_2 \vdash t_2 : T_2$ **by** *simp*
**with** *tT* **have** *ts*$_1$: $\downarrow_e \|\Delta_1\|\ 0\ (\Gamma_1 \mathbin{@} \uparrow_e \|\Delta_1\|\ 0\ \Delta_2) \mathbin{@} \Gamma_2 \vdash$
$t_2[\|\Gamma_1 \mathbin{@} \uparrow_e \|\Delta_1\|\ 0\ \Delta_2\| \mapsto_s ts_1] : \downarrow_\tau \|\Delta_1\|\ \|\Gamma_1 \mathbin{@} \uparrow_e \|\Delta_1\|\ 0\ \Delta_2\|\ T_2$
**using** *p* **by** (*rule P-Cons*)
**from** *fps′ P-Cons(5)* **have** $\vdash$ *fps* [$\triangleright$] *fs′* $\Rightarrow$ *ts*$_2$ **by** (*rule matchs-tl*)
**with** *fs′ ts*$_1$ [*simplified*]
**have** $\downarrow_e \|\Delta_2\|\ 0\ (\downarrow_e \|\Delta_1\|\ \|\Delta_2\|\ \Gamma_1) \mathbin{@} \Gamma_2 \vdash t_2[\|\Gamma_1\| + \|\Delta_2\| \mapsto_s ts_1][\|\downarrow_e \|\Delta_1\|\ \|\Delta_2\|\ \Gamma_1\| \mapsto_s ts_2] :$
$\downarrow_\tau \|\Delta_2\|\ \|\downarrow_e \|\Delta_1\|\ \|\Delta_2\|\ \Gamma_1\|\ (\downarrow_\tau \|\Delta_1\|\ (\|\Gamma_1\| + \|\Delta_2\|)\ T_2)$
**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* $\Gamma$ *i U T t′*)
**from** ‹*Var i* $\longmapsto$ *t′*›
**show** *?case* **by** *cases*
**next**
**case** (*T-Abs* $T_1$ $\Gamma$ $t_2$ $T_2$ *t′*)
**from** ‹($\lambda$:$T_1$. $t_2$) $\longmapsto$ *t′*›
**show** *?case* **by** *cases*
**next**
**case** (*T-App* $\Gamma$ $t_1$ $T_{11}$ $T_{12}$ $t_2$ *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** $\Delta$=[], *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 ‹t₁ ⟶ t''›
    have Γ ⊢ t'' : T₁₁ → T₁₂ by (rule T-App)
    hence Γ ⊢ t'' • t₂ : T₁₂ using ‹Γ ⊢ t₂ : T₁₁›
      by (rule typing-typings.T-App)
    with E-App1 show ?thesis by simp
  next
    case (E-App2 t'')
    from ‹t₂ ⟶ t''›
    have Γ ⊢ t'' : T₁₁ by (rule T-App)
    with T-App(1) have Γ ⊢ t₁ • t'' : T₁₂
      by (rule typing-typings.T-App)
    with E-App2 show ?thesis by simp
  qed
next
  case (T-TAbs T₁ Γ t₂ T₂ t')
  from ‹(λ<:T₁. t₂) ⟶ t'›
  show ?case by cases
next
  case (T-TApp Γ t₁ T₁₁ T₁₂ T₂ t')
  from ‹t₁ •τ T₂ ⟶ t'›
  show ?case
  proof cases
    case (E-TAbs T₁₁' t₁₂)
    with T-TApp have Γ ⊢ (λ<:T₁₁'. t₁₂) : (∀<:T₁₁. T₁₂) by simp
    then obtain S'
      where TVarB T₁₁ :: Γ ⊢ t₁₂ : S'
      and TVarB T₁₁ :: Γ ⊢ S' <: T₁₂ by (rule TAbs-type') blast
    hence TVarB T₁₁ :: Γ ⊢ t₁₂ : T₁₂ by (rule T-Sub)
    hence Γ ⊢ t₁₂[0 ↦τ T₂] : T₁₂[0 ↦τ T₂]τ using T-TApp(3)
      by (rule substT-type [where Δ=[], simplified])
    with E-TAbs show ?thesis by simp
  next
    case (E-TApp t'')
    from ‹t₁ ⟶ t''›
    have Γ ⊢ t'' : (∀<:T₁₁. T₁₂) by (rule T-TApp)
    hence Γ ⊢ t'' •τ T₂ : T₁₂[0 ↦τ T₂]τ using ‹Γ ⊢ T₂ <: T₁₁›
      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 ⟶ t'›
  have Γ ⊢ t' : S by (rule T-Sub)
  then show ?case using ‹Γ ⊢ S <: T›
    by (rule typing-typings.T-Sub)
next
```

```
  case (T-Let Γ t₁ T₁ p Δ t₂ T₂ t')
  from ‹(LET p = t₁ IN t₂) ⟼ t'›
  show ?case
  proof cases
    case (E-LetV ts)
    from T-Let (3,1,4) ‹⊢ p ▷ t₁ ⇒ ts›
    have Γ ⊢ t₂[0 ↦ₛ ts] : ↓τ ‖Δ‖ 0 T₂
      by (rule match-type(1) [of - - - - - [], simplified])
    with E-LetV show ?thesis by simp
  next
    case (E-Let t'')
    from ‹t₁ ⟼ t''›
    have Γ ⊢ t'' : T₁ by (rule T-Let)
    hence Γ ⊢ (LET p = t'' IN t₂) : ↓τ ‖Δ‖ 0 T₂ 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 ⟼ t'›
  obtain fs' where t': t' = Rcd fs' and fs: fs [⟼] fs'
    by cases simp-all
  from fs have Γ ⊢ fs' [:] fTs by (rule T-Rcd)
  hence Γ ⊢ 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 ⟼ t'›
  show ?case
  proof cases
    case (E-ProjRcd fs)
    with T-Proj have Γ ⊢ Rcd fs : RcdT fTs by simp
    hence ∀(l, U)∈set fTs. ∃ u. fs⟨l⟩? = ⌊u⌋ ∧ Γ ⊢ 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 ⟼ t''›
    have Γ ⊢ t'' : RcdT fTs by (rule T-Proj)
    hence Γ ⊢ 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 ‹[] [⟼] 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 ⟼ t'›
    have Γ ⊢ t' : T by (rule T-Cons)
    hence Γ ⊢ (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 Γ ⊢ fs'' [:] fTs by (rule T-Cons)
    moreover from fs T-Cons have fs''⟨l⟩? = ⊥ by simp
    ultimately have Γ ⊢ (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: [] ⊢ v : T₁ → T₂
  shows v ∈ value ⟹ ∃ t S. v = (λ:S. t) using ty
proof (induct []::env v T₁ → T₂ arbitrary: T₁ T₂ rule: typing-induct)
  case T-Abs
  show ?case by iprover
next
  case (T-App t₁ T₁₁ t₂ T₁ T₂)
  from ‹t₁ · t₂ ∈ value›
  show ?case by cases
next
  case (T-TApp t₁ T₁₁ T₁₂ T₂ T₁ T₂')
  from ‹t₁ ·τ T₂ ∈ value›
  show ?case by cases
next
  case (T-Sub t S T₁ T₂)
  from ‹[] ⊢ S <: T₁ → T₂›
  obtain S₁ S₂ where S: S = S₁ → S₂
    by cases (auto simp add: T-Sub)
  show ?case by (rule T-Sub S)+
next
  case (T-Let t₁ T₁ p Δ t₂ T₂ T₁' T₂')
  from ‹(LET p = t₁ IN t₂) ∈ value›
  show ?case by cases
next
  case (T-Proj t fTs l T₁ T₂)
  from ‹t..l ∈ 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-induct*)
**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*)+
**next**
**case** (*T-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-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-induct*)
**case** (*T-App* $t_1\ T_{11}\ t_2\ fTs$)
**from** ‹$t_1 \cdot t_2 \in value$›
**show** *?case* **by** *cases*
**next**
**case** (*T-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 t₁ T₁ p Δ t₂ T₂ fTs)
  from ‹(LET p = t₁ IN t₂) ∈ value›
  show ?case by cases
next
  case (T-Rcd fs fTs)
  from ‹Rcd fs ∈ value›
  show ?case using T-Rcd by cases simp-all
next
  case (T-Proj t fTs l fTs')
  from ‹t..l ∈ value›
  show ?case by cases
qed simp-all

theorem reorder-prop:
  ∀(l, t) ∈ set fs. P t ⟹ ∀(l, U)∈set fTs. ∃u. fs⟨l⟩? = ⌊u⌋ ⟹
    ∀(l, t) ∈ set (map (λ(l, T). (l, the (fs⟨l⟩?))) fTs). P t
  apply (induct fs)
  apply simp
  apply (simp add: split-paired-all)
  apply simp
  apply (rule ballI)
  apply (simp add: split-paired-all)
  apply (drule bpspec)
  apply assumption
  apply (erule exE)
  apply (simp split: if-split-asm)
  apply (auto dest: assoc-set)
  done
```

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:
  ⊢ p : T ⇒ Δ ⟹ [] ⊢ t : T ⟹ t ∈ value ⟹
    ∃ts. ⊢ p ▷ t ⇒ ts
  ⊢ fps [:] fTs ⇒ Δ ⟹ [] ⊢ fs [:] fTs ⟹
    ∀(l, t) ∈ set fs. t ∈ value ⟹ ∃us. ⊢ fps [▷] fs ⇒ 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** $\exists us.\ \vdash fps\ [\triangleright]\ map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs \Rightarrow us$
**by** (*rule P-Rcd*)
**then obtain** *us* **where** $\vdash fps\ [\triangleright]\ map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs \Rightarrow us$ **..**
**with** *P-Rcd*(*1*) *assoc* **have** $\vdash fps\ [\triangleright]\ fs \Rightarrow us$ **by** (*rule matchs-reorder'*)
**hence** $\vdash PRcd\ fps \triangleright Rcd\ fs \Rightarrow us$ **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* $\Delta_1$ *fps fTs* $\Delta_2$ *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** $\exists ts.\ \vdash p \triangleright t \Rightarrow ts$ **by** (*rule P-Cons*)
**then obtain** *ts* **where** $\vdash p \triangleright t \Rightarrow ts$ **..**
**moreover from** *P-Cons fs* **have** $\forall (l,\ t) \in set\ fs'.\ t \in value$ **by** *auto*
**with** *fs'* **have** $\exists us.\ \vdash fps\ [\triangleright]\ fs' \Rightarrow us$ **by** (*rule P-Cons*)
**then obtain** *us* **where** $\vdash fps\ [\triangleright]\ fs' \Rightarrow us$ **..**
**hence** $\vdash fps\ [\triangleright]\ (l,\ t) :: fs' \Rightarrow us$ **using** *P-Cons*(*5*) **by** (*rule matchs-mono*)
**ultimately have** $\vdash (l,\ p) :: fps\ [\triangleright]\ (l,\ t) :: fs' \Rightarrow ts\ @\ us$
**by** (*rule M-Cons*)
**with** *fs* **show** *?case* **by** *iprover*
**qed**

**theorem** *progress*: — A.16
$[] \vdash t : T \Longrightarrow t \in value \vee (\exists t'.\ t \longmapsto t')$
$[] \vdash fs\ [:]\ fTs \Longrightarrow (\forall (l,\ t) \in set\ fs.\ t \in value) \vee (\exists fs'.\ fs\ [\longmapsto]\ fs')$
**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** $t_1 \in value \vee (\exists t'.\ t_1 \longmapsto t')$ **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-evals.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-evals.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-evals.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-evals.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-evals.intros)
    thus ?thesis by iprover
  qed
next
  case (T-Sub t S T)
  show ?case by (rule T-Sub)
next
  case (T-Let t₁ T₁ p Δ t₂ T₂)
  hence t₁ ∈ value ∨ (∃ t'. t₁ ⟶ t') by simp
```

```
  thus ?case
  proof
    assume t₁: t₁ ∈ value
    with T-Let have ∃ ts. ⊢ p ▷ t₁ ⇒ ts
      by (auto intro: ptyping-match)
    with t₁ show ?thesis by (blast intro: eval-evals.intros)
  next
    assume ∃ t'. t₁ ⟼ 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 ∈ value ∨ (∃ t'. t ⟼ t') by simp
  thus ?case
  proof
    assume tv: t ∈ value
    with T-Proj obtain fs where
      t: t = Rcd fs and fs: ∀ (l, t) ∈ set fs. t ∈ value
      by (auto dest: RcdT-canonical)
    with T-Proj have [] ⊢ Rcd fs : RcdT fTs by simp
    hence ∀ (l, U)∈set fTs. ∃ u. fs⟨l⟩? = ⌊u⌋ ∧ [] ⊢ u : U
      by (rule Rcd-type1')
    with T-Proj obtain u where u: fs⟨l⟩? = ⌊u⌋ by (blast dest: assoc-set)
    with fs have u ∈ value by (blast dest: assoc-set)
    with u t show ?case by (blast intro: eval-evals.intros)
  next
    assume ∃ t'. t ⟼ 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* ⇒ *trm*) *set*
  **and** *rctxt* :: (*trm* ⇒ *rcd*) *set*
**where**
  *C-Hole*: (λ*t*. *t*) ∈ *ctxt*
| *C-App1*: *E* ∈ *ctxt* ⟹ (λ*t*. *E t* ∙ *u*) ∈ *ctxt*
| *C-App2*: *v* ∈ *value* ⟹ *E* ∈ *ctxt* ⟹ (λ*t*. *v* ∙ *E t*) ∈ *ctxt*
| *C-TApp*: *E* ∈ *ctxt* ⟹ (λ*t*. *E t* ∙$_\tau$ *T*) ∈ *ctxt*
| *C-Proj*: *E* ∈ *ctxt* ⟹ (λ*t*. *E t*..*l*) ∈ *ctxt*
| *C-Rcd*: *E* ∈ *rctxt* ⟹ (λ*t*. *Rcd* (*E t*)) ∈ *ctxt*
| *C-Let*: *E* ∈ *ctxt* ⟹ (λ*t*. *LET p* = *E t IN u*) ∈ *ctxt*
| *C-hd*: *E* ∈ *ctxt* ⟹ (λ*t*. (*l*, *E t*) :: *fs*) ∈ *rctxt*
| *C-tl*: *v* ∈ *value* ⟹ *E* ∈ *rctxt* ⟹ (λ*t*. (*l*, *v*) :: *E t*) ∈ *rctxt*

**lemmas** *rctxt-induct* = *ctxt-rctxt.inducts*(*2*)
  [*of* - λ*x*. *True*, *simplified True-simps*, *consumes 1*, *case-names C-hd C-tl*]

**lemma** *rctxt-labels*:
  **assumes** *H*: *E* ∈ *rctxt*
  **shows** *E t*⟨*l*⟩$_?$ = ⊥ ⟹ *E t'*⟨*l*⟩$_?$ = ⊥ **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* ⇒ *trm* ⇒ *bool*  (**infixl** ⟼$_c$ *50*)
**where**
  *E-Ctxt*: *t* ⟼$_c$ *t'* ⟹ *E* ∈ *ctxt* ⟹ *E t* ⟼$_c$ *E t'*
| *E-Abs*: *v*$_2$ ∈ *value* ⟹ (λ:*T*$_{11}$. *t*$_{12}$) ∙ *v*$_2$ ⟼$_c$ *t*$_{12}$[*0* ↦ *v*$_2$]
| *E-TAbs*: (λ<:*T*$_{11}$. *t*$_{12}$) ∙$_\tau$ *T*$_2$ ⟼$_c$ *t*$_{12}$[*0* ↦$_\tau$ *T*$_2$]
| *E-LetV*: *v* ∈ *value* ⟹ ⊢ *p* ▷ *v* ⇒ *ts* ⟹ (*LET p* = *v IN t*) ⟼$_c$ *t*[*0* ↦$_s$ *ts*]
| *E-ProjRcd*: *fs*⟨*l*⟩$_?$ = ⌊*v*⌋ ⟹ *v* ∈ *value* ⟹ *Rcd fs*..*l* ⟼$_c$ *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
  Γ ⊢ *u* : *T* ⟹ *E* ∈ *ctxt* ⟹ *u* = *E t* ⟹
    (⋀*T*$_0$. Γ ⊢ *t* : *T*$_0$ ⟹ Γ ⊢ *t'* : *T*$_0$) ⟹ Γ ⊢ *E t'* : *T*
  Γ ⊢ *fs* [:] *fTs* ⟹ *E*$_r$ ∈ *rctxt* ⟹ *fs* = *E*$_r$ *t* ⟹
    (⋀*T*$_0$. Γ ⊢ *t* : *T*$_0$ ⟹ Γ ⊢ *t'* : *T*$_0$) ⟹ Γ ⊢ *E*$_r$ *t'* [:] *fTs*
**proof** (*induct arbitrary*: *E t t'* **and** *E*$_r$ *t t'* *set*: *typing typings*)

**case** (*T-Var* Γ *i U T E t t′*)
**from** ‹*E* ∈ *ctxt*›
**have** *E* = (λ*t*. *t*) **using** *T-Var* **by** *cases simp-all*
**with** *T-Var* **show** *?case* **by** (*blast intro*: *typing-typings.intros*)
**next**
**case** (*T-Abs* $T_1$ $T_2$ Γ $t_2$ *E t t′*)
**from** ‹*E* ∈ *ctxt*›
**have** *E* = (λ*t*. *t*) **using** *T-Abs* **by** *cases simp-all*
**with** *T-Abs* **show** *?case* **by** (*blast intro*: *typing-typings.intros*)
**next**
**case** (*T-App* Γ $t_1$ $T_{11}$ $T_{12}$ $t_2$ *E t t′*)
**from** ‹*E* ∈ *ctxt*›
**show** *?case* **using** *T-App*
**by** *cases* (*simp-all*, (*blast intro*: *typing-typings.intros*)+)
**next**
**case** (*T-TAbs* $T_1$ Γ $t_2$ $T_2$ *E t t′*)
**from** ‹*E* ∈ *ctxt*›
**have** *E* = (λ*t*. *t*) **using** *T-TAbs* **by** *cases simp-all*
**with** *T-TAbs* **show** *?case* **by** (*blast intro*: *typing-typings.intros*)
**next**
**case** (*T-TApp* Γ $t_1$ $T_{11}$ $T_{12}$ $T_2$ *E t t′*)
**from** ‹*E* ∈ *ctxt*›
**show** *?case* **using** *T-TApp*
**by** *cases* (*simp-all*, (*blast intro*: *typing-typings.intros*)+)
**next**
**case** (*T-Sub* Γ *t S T E ta t′*)
**thus** *?case* **by** (*blast intro*: *typing-typings.intros*)
**next**
**case** (*T-Let* Γ $t_1$ $T_1$ *p* Δ $t_2$ $T_2$ *E t t′*)
**from** ‹*E* ∈ *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* ∈ *ctxt*›
**show** *?case* **using** *T-Rcd*
**by** *cases* (*simp-all*, (*blast intro*: *typing-typings.intros*)+)
**next**
**case** (*T-Proj* Γ *t fTs l T E ta t′*)
**from** ‹*E* ∈ *ctxt*›
**show** *?case* **using** *T-Proj*
**by** *cases* (*simp-all*, (*blast intro*: *typing-typings.intros*)+)
**next**
**case** (*T-Nil* Γ *E t t′*)
**from** ‹*E* ∈ *rctxt*›
**show** *?case* **using** *T-Nil*
**by** *cases simp-all*
**next**
**case** (*T-Cons* Γ *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* $\Gamma$ $(\lambda{:}T_{11}.\ t_{12}) \cdot t_2$ $T$ *arbitrary*: $T_{11}$ $t_{12}$ $t_2$ *rule*: *typing-induct*)
**case** (*T-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** $\Delta$=[], *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* $\Gamma$ $(\lambda{<:}T_{11}.\ t_{12}) \cdot_\tau T_2$ $T$ *arbitrary*: $T_{11}$ $t_{12}$ $T_2$ *rule*: *typing-induct*)
**case** (*T-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** $\Delta$=[], *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* $\Gamma$ *LET* $p = t_1$ *IN* $t_2$ $T$ *arbitrary*: $p$ $t_1$ $t_2$ *ts rule*: *typing-induct*)
**case** (*T-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*⟨*l*⟩$_?$ = ⌊*v*⌋ $\Longrightarrow \Gamma \vdash v : T$
**using** *H*
**proof** (*induct* $\Gamma$ *Rcd fs..l* $T$ *arbitrary*: *fs l v rule*: *typing-induct*)
**case** (*T-Proj* $\Gamma$ *fTs l T fs v*)
**from** ‹$\Gamma \vdash$ *Rcd fs* : *RcdT fTs*›
**have** $\forall (l,\ U) \in$ *set fTs*. $\exists u$. *fs*⟨*l*⟩$_?$ = ⌊*u*⌋ $\wedge \Gamma \vdash u : U$
**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*: $\Gamma$ $T$)
**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 t₁ ∈ value
    with T-TApp obtain t S where t₁ = (λ<:S. t)
      by (auto dest!: TyAll-canonical)
    hence t₁ ·_τ T₂ ⟼_c t[0 ↦_τ T₂] by simp (rule eval.intros)
    thus ?thesis by (iprover intro: C-Hole)
  next
    assume ∃ E t₀ t₀'. E ∈ ctxt ∧ t₁ = E t₀ ∧ t₀ ⟼_c t₀'
    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 t₁ T₁ p Δ t₂ T₂)
  from ‹t₁ ∈ value ∨ (∃ E t₀ t₀'. E ∈ ctxt ∧ t₁ = E t₀ ∧ t₀ ⟼_c t₀')›
  show ?case
  proof
    assume t₁: t₁ ∈ value
    with T-Let have ∃ ts. ⊢ p ▷ t₁ ⇒ ts
      by (auto intro: ptyping-match)
    with t₁ show ?thesis by (iprover intro: eval.intros C-Hole)
  next
    assume ∃ E t₀ t₀'. E ∈ ctxt ∧ t₁ = E t₀ ∧ t₀ ⟼_c t₀'
    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 ∈ value ∨ (∃ E t₀ t₀'. E ∈ ctxt ∧ t = E t₀ ∧ t₀ ⟼_c t₀')›
  show ?case
  proof
    assume tv: t ∈ value
    with T-Proj obtain fs where
      t: t = Rcd fs and fs: ∀ (l, t) ∈ set fs. t ∈ value
      by (auto dest: RcdT-canonical)
    with T-Proj have [] ⊢ Rcd fs : RcdT fTs by simp
    hence ∀ (l, U)∈set fTs. ∃ u. fs⟨l⟩? = ⌊u⌋ ∧ [] ⊢ u : U
      by (rule Rcd-type1')
    with T-Proj obtain u where u: fs⟨l⟩? = ⌊u⌋ by (blast dest: assoc-set)
    with fs have u ∈ value by (blast dest: assoc-set)
    with u t show ?thesis by (iprover intro: eval.intros C-Hole)
  next
    assume ∃ E t₀ t₀'. E ∈ ctxt ∧ t = E t₀ ∧ t₀ ⟼_c t₀'
    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: [] ⊢ t : T
  shows t ∈ value ∨ (∃ t'. t ⟼c t')
proof −
  from H have t ∈ value ∨ (∃ E t0 t0'. E ∈ ctxt ∧ t = E t0 ∧ t0 ⟼c t0')
    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 ∈ ctxt ⟹ t ⟼ t' ⟹ E t ⟼ E t'
  E_r ∈ rctxt ⟹ t ⟼ t' ⟹ E_r t [⟼] E_r t'
  by (induct rule: ctxt-rctxt.inducts) (auto intro: eval-evals.intros)

lemma eval-evalc-eq: (t ⟼ t') = (t ⟼c t')
proof
  fix ts ts'
  have r: t ⟼ t' ⟹ t ⟼c t' and
    ts [⟼] ts' ⟹ ∃ E t t'. E ∈ rctxt ∧ ts = E t ∧ ts' = E t' ∧ t ⟼c t'
    by (induct rule: eval-evals.inducts) (iprover intro: ctxt-rctxt.intros eval.intros)+
  assume t ⟼ t'
  thus t ⟼c t' by (rule r)
next
  assume t ⟼c t'
  thus t ⟼ 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* ≡ ∀ <:*Top*. (∀ <:*TVar 0*. (∀ <:*TVar 1*. (*TVar 2* → *TVar 1*) → *TVar 0* → *TVar 1*))

**definition**
*fact2* :: *trm* **where**
*fact2* ≡
*LET PVar natT* =
(λ<:*Top*. λ<:*TVar 0*. λ<:*TVar 1*. λ:*TVar 2* → *TVar 1*. λ: *TVar 1*. *Var 1* · *Var 0*)
*IN*
*LET PRcd*
[(''*pluspp*'', *PVar* (*natT* → *natT* → *natT*)),
(''*multpp*'', *PVar* (*natT* → *natT* → *natT*))] = *Rcd*
[(''*multpp*'', λ:*natT*. λ:*natT*. λ<:*Top*. λ<:*TVar 0*. λ<:*TVar 1*. λ:*TVar 2* → *TVar 1*.
*Var 5* ·$_\tau$ *TVar 3* ·$_\tau$ *TVar 2* ·$_\tau$ *TVar 1* · (*Var 4* ·$_\tau$ *TVar 3* ·$_\tau$ *TVar 2* ·$_\tau$ *TVar 1*) · *Var 0*),
(''*pluspp*'', λ:*natT*. λ:*natT*. λ<:*Top*. λ<:*TVar 0*. λ<:*TVar 1*. λ:*TVar 2* → *TVar 1*. λ:*TVar 1*.
*Var 6* ·$_\tau$ *TVar 4* ·$_\tau$ *TVar 3* ·$_\tau$ *TVar 3* · *Var 1* ·
(*Var 5* ·$_\tau$ *TVar 4* ·$_\tau$ *TVar 3* ·$_\tau$ *TVar 2* · *Var 1* · *Var 0*))]
*IN*
*Var 0* · (*Var 1* · *Var 2* · *Var 2*) · *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.