

A Solution to the POPLMARK Challenge in Isabelle/HOL

Stefan Berghofer

June 16, 2019

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	2
2	Formalization of the basic calculus	4
2.1	Types and Terms	4
2.2	Lifting and Substitution	5
2.3	Well-formedness	9
2.4	Subtyping	13
2.5	Typing	21
2.6	Evaluation	28
3	Extending the calculus with records	33
3.1	Types and Terms	33
3.2	Lifting and Substitution	35
3.3	Well-formedness	44
3.4	Subtyping	49
3.5	Typing	59
3.6	Evaluation	71
4	Evaluation contexts	83
5	Executing the specification	90

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  $\implies$  PROP P)  $\equiv$  PROP P
(PROP P  $\implies$  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*: $(\bigwedge x y. (x, y) \in A \implies P x y) \implies \forall (x, y) \in A. P x y$
by *blast*

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

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

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

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

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

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

notation

Some ($\lfloor _ \rfloor$)

notation*None* (\perp)**notation***length* ($\|-\|$)**notation***Cons* ($- :: / - [66, 65] 65$)

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

primrec*nth-el* :: 'a list \Rightarrow nat \Rightarrow 'a option $(-\langle-\rangle [90, 0] 91)$ **where** $\square\langle i \rangle = \perp$ $| (x \# xs)\langle i \rangle = (\text{case } i \text{ of } 0 \Rightarrow [x] \mid \text{Suc } j \Rightarrow xs \langle j \rangle)$ **lemma** [*simp*]: $i < \|xs\| \Longrightarrow (xs @ ys)\langle i \rangle = xs\langle i \rangle$ **apply** (*induct xs arbitrary: i*)**apply** *simp***apply** (*case-tac i*)**apply** *simp-all***done****lemma** [*simp*]: $\|xs\| \leq i \Longrightarrow (xs @ ys)\langle i \rangle = ys\langle i - \|xs\| \rangle$ **apply** (*induct xs arbitrary: i*)**apply** *simp***apply** (*case-tac i*)**apply** *simp-all***done**

Association lists

primrec *assoc* :: ('a \times 'b) list \Rightarrow 'a \Rightarrow 'b option $(-\langle-\rangle? [90, 0] 91)$ **where** $\square\langle a \rangle? = \perp$ $| (x \# xs)\langle a \rangle? = (\text{if } \text{fst } x = a \text{ then } [\text{snd } x] \text{ else } xs\langle a \rangle?)$ **primrec** *unique* :: ('a \times 'b) list \Rightarrow bool**where***unique* $\square = \text{True}$ $| \text{unique } (x \# xs) = (xs\langle \text{fst } x \rangle? = \perp \wedge \text{unique } xs)$ **lemma** *assoc-set*: $ps\langle x \rangle? = [y] \Longrightarrow (x, y) \in \text{set } ps$ **by** (*induct ps*) (*auto split: if-split-asm*)**lemma** *map-assoc-None* [*simp*]: $ps\langle x \rangle? = \perp \Longrightarrow \text{map } (\lambda(x, y). (x, f x y)) ps\langle x \rangle? = \perp$ **by** (*induct ps*) *auto*

no-syntax

$-Map :: maplets => 'a \rightarrow 'b \ ((1[-]))$

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  $\rightarrow$  200)  
  | TyAll type type (( $\exists \forall <:-./ -$ ) [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  $\Rightarrow$  bool
```

where

```
  is-TVarB (VarB T) = False  
  | is-TVarB (TVarB T) = True
```

```
primrec type-ofB :: binding  $\Rightarrow$  type
```

where

```
  type-ofB (VarB T) = T  
  | type-ofB (TVarB T) = T
```

```
primrec mapB :: (type  $\Rightarrow$  type)  $\Rightarrow$  binding  $\Rightarrow$  binding
```

where

```
  mapB f (VarB T) = VarB (f T)
```

| $\text{mapB } f \text{ (TVarB } T) = \text{TVarB } (f \ T)$

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

```
datatype trm =
  Var nat
| Abs type trm (( $\exists \lambda$ :-./ -) [0, 10] 10)
| TAbs type trm (( $\exists \lambda$ <:-./ -) [0, 10] 10)
| App trm trm (infixl  $\cdot$  200)
| TApp trm type (infixl  $\cdot_{\tau}$  200)
```

2.2 Lifting and Substitution

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

```
primrec liftT :: nat  $\Rightarrow$  nat  $\Rightarrow$  type  $\Rightarrow$  type ( $\uparrow_{\tau}$ )
where
   $\uparrow_{\tau} n k \text{ (TVar } i) = (\text{if } i < k \text{ then TVar } i \text{ else TVar } (i + n))$ 
|  $\uparrow_{\tau} n k \text{ Top} = \text{Top}$ 
|  $\uparrow_{\tau} n k \text{ (} T \rightarrow U) = \uparrow_{\tau} n k T \rightarrow \uparrow_{\tau} n k U$ 
|  $\uparrow_{\tau} n k \text{ (}\forall <: T. U) = (\forall <: \uparrow_{\tau} n k T. \uparrow_{\tau} n (k + 1) U)$ 
```

```
primrec lift :: nat  $\Rightarrow$  nat  $\Rightarrow$  trm  $\Rightarrow$  trm ( $\uparrow$ )
where
   $\uparrow n k \text{ (Var } i) = (\text{if } i < k \text{ then Var } i \text{ else Var } (i + n))$ 
|  $\uparrow n k \text{ (}\lambda:T. t) = (\lambda:\uparrow_{\tau} n k T. \uparrow n (k + 1) t)$ 
|  $\uparrow n k \text{ (}\lambda <: T. t) = (\lambda <: \uparrow_{\tau} n k T. \uparrow n (k + 1) t)$ 
|  $\uparrow n k \text{ (} s \cdot t) = \uparrow n k s \cdot \uparrow n k t$ 
|  $\uparrow n k \text{ (} t \cdot_{\tau} T) = \uparrow n k t \cdot_{\tau} \uparrow_{\tau} n k T$ 
```

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

```
primrec substTT :: type  $\Rightarrow$  nat  $\Rightarrow$  type  $\Rightarrow$  type ( $[- \mapsto_{\tau} -]_{\tau}$  [300, 0, 0] 300)
where
   $(\text{TVar } i)[k \mapsto_{\tau} S]_{\tau} =$ 
     $(\text{if } k < i \text{ then TVar } (i - 1) \text{ else if } i = k \text{ then } \uparrow_{\tau} k \ 0 \ S \text{ else TVar } i)$ 
|  $\text{Top}[k \mapsto_{\tau} S]_{\tau} = \text{Top}$ 
|  $(T \rightarrow U)[k \mapsto_{\tau} S]_{\tau} = T[k \mapsto_{\tau} S]_{\tau} \rightarrow U[k \mapsto_{\tau} S]_{\tau}$ 
|  $(\forall <: T. U)[k \mapsto_{\tau} S]_{\tau} = (\forall <: T[k \mapsto_{\tau} S]_{\tau}. U[k+1 \mapsto_{\tau} S]_{\tau})$ 
```

primrec $decT :: nat \Rightarrow nat \Rightarrow type \Rightarrow type \ (\downarrow_\tau)$

where

$\downarrow_\tau \ 0 \ k \ T = T$
 $|\ \downarrow_\tau \ (Suc \ n) \ k \ T = \downarrow_\tau \ n \ k \ (T[k \mapsto_\tau \ Top]_\tau)$

primrec $subst :: trm \Rightarrow nat \Rightarrow trm \Rightarrow trm \ (-[- \mapsto -] \ [300, 0, 0] \ 300)$

where

$(Var \ i)[k \mapsto s] = (if \ k < i \ then \ Var \ (i - 1) \ else \ if \ i = k \ then \ \uparrow \ k \ 0 \ s \ else \ Var \ i)$
 $|\ (t \cdot u)[k \mapsto s] = t[k \mapsto s] \cdot u[k \mapsto s]$
 $|\ (t \cdot_\tau \ T)[k \mapsto s] = t[k \mapsto s] \cdot_\tau \ \downarrow_\tau \ 1 \ k \ T$
 $|\ (\lambda:T. \ t)[k \mapsto s] = (\lambda:\downarrow_\tau \ 1 \ k \ T. \ t[k+1 \mapsto s])$
 $|\ (\lambda<:T. \ t)[k \mapsto s] = (\lambda<:\downarrow_\tau \ 1 \ k \ T. \ t[k+1 \mapsto s])$

primrec $substT :: trm \Rightarrow nat \Rightarrow type \Rightarrow trm \ (-[- \mapsto_\tau -] \ [300, 0, 0] \ 300)$

where

$(Var \ i)[k \mapsto_\tau \ S] = (if \ k < i \ then \ Var \ (i - 1) \ else \ Var \ i)$
 $|\ (t \cdot u)[k \mapsto_\tau \ S] = t[k \mapsto_\tau \ S] \cdot u[k \mapsto_\tau \ S]$
 $|\ (t \cdot_\tau \ T)[k \mapsto_\tau \ S] = t[k \mapsto_\tau \ S] \cdot_\tau \ T[k \mapsto_\tau \ S]_\tau$
 $|\ (\lambda:T. \ t)[k \mapsto_\tau \ S] = (\lambda:T[k \mapsto_\tau \ S]_\tau. \ t[k+1 \mapsto_\tau \ S])$
 $|\ (\lambda<:T. \ t)[k \mapsto_\tau \ S] = (\lambda<:T[k \mapsto_\tau \ S]_\tau. \ t[k+1 \mapsto_\tau \ S])$

Lifting and substitution extends to typing contexts as follows:

primrec $liftE :: nat \Rightarrow nat \Rightarrow env \Rightarrow env \ (\uparrow_e)$

where

$\uparrow_e \ n \ k \ [] = []$
 $|\ \uparrow_e \ n \ k \ (B :: \Gamma) = mapB \ (\uparrow_\tau \ n \ (k + \|\Gamma\|)) \ B :: \uparrow_e \ n \ k \ \Gamma$

primrec $substE :: env \Rightarrow nat \Rightarrow type \Rightarrow env \ (-[- \mapsto_\tau -]_e \ [300, 0, 0] \ 300)$

where

$[][k \mapsto_\tau \ T]_e = []$
 $|\ (B :: \Gamma)[k \mapsto_\tau \ T]_e = mapB \ (\lambda U. \ U[k + \|\Gamma\| \mapsto_\tau \ T]_\tau) \ B :: \Gamma[k \mapsto_\tau \ T]_e$

primrec $decE :: nat \Rightarrow nat \Rightarrow env \Rightarrow env \ (\downarrow_e)$

where

$\downarrow_e \ 0 \ k \ \Gamma = \Gamma$
 $|\ \downarrow_e \ (Suc \ n) \ k \ \Gamma = \downarrow_e \ n \ k \ (\Gamma[k \mapsto_\tau \ Top]_e)$

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

lemma $liftE\text{-length} \ [simp]: \ \|\uparrow_e \ n \ k \ \Gamma\| = \|\Gamma\|$

by $(induct \ \Gamma) \ simp\text{-all}$

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

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

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

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

lemma *liftT-liftT'* [simp]:
 $i + m \leq j \implies \uparrow_\tau n j (\uparrow_\tau m i T) = \uparrow_\tau m i (\uparrow_\tau n (j - m) T)$
 apply (induct T arbitrary: $i j m n$)
 apply simp-all
 apply arith
 apply (subgoal-tac $\text{Suc } j - m = \text{Suc } (j - m)$)
 apply simp
 apply arith
 done

lemma *lift-size* [simp]: $\text{size } (\uparrow_\tau n k T) = \text{size } T$
 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]:
 $k \leq k' \implies k' < k + n \implies (\uparrow_\tau n k T)[k' \mapsto_\tau U]_\tau = \uparrow_\tau (n - 1) k T$

by (induct T arbitrary: k k') simp-all

theorem liftT-substT [simp]:

$k \leq k' \implies \uparrow_\tau n k (T[k' \mapsto_\tau U]_\tau) = \uparrow_\tau n k T[k' + n \mapsto_\tau U]_\tau$

apply (induct T arbitrary: k k')

apply simp-all

done

theorem liftT-substT' [simp]: $k' < k \implies$

$\uparrow_\tau n k (T[k' \mapsto_\tau U]_\tau) = \uparrow_\tau n (k + 1) T[k' \mapsto_\tau \uparrow_\tau n (k - k') U]_\tau$

apply (induct T arbitrary: k k')

apply simp-all

apply arith

done

lemma liftT-substT-Top [simp]:

$k \leq k' \implies \uparrow_\tau n k' (T[k \mapsto_\tau Top]_\tau) = \uparrow_\tau n (Suc k') T[k \mapsto_\tau Top]_\tau$

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 $\wedge x. PROP P x$ for $P :: - \implies prop$)

apply (drule-tac $x=n$ in meta-spec)

apply (drule-tac $x=Suc k$ in meta-spec)

apply simp

done

lemma lift-lift [simp]:

$k \leq k' \implies k' \leq k + n \implies \uparrow n' k' (\uparrow n k t) = \uparrow (n + n') k t$

by (induct t arbitrary: k k') simp-all

lemma substT-substT:

$i \leq j \implies T[Suc j \mapsto_\tau V]_\tau [i \mapsto_\tau U [j - i \mapsto_\tau V]_\tau]_\tau = T[i \mapsto_\tau U]_\tau [j \mapsto_\tau V]_\tau$

apply (induct T arbitrary: i j U V)

apply (simp-all add: diff-Suc split: nat.split)

apply (thin-tac $\wedge x. PROP P x$ for $P :: - \implies prop$)

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

inductive

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

where

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

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

inductive

$well\text{-}formedE :: env \Rightarrow bool\ (- \vdash_{wf} [50] 50)$

and $well\text{-}formedB :: env \Rightarrow binding \Rightarrow bool\ (- \vdash_{wfB} - [50, 50] 50)$

where

$\Gamma \vdash_{wfB}\ B \equiv \Gamma \vdash_{wf}\ type\text{-}ofB\ B$
 $| wf\text{-}Nil: [] \vdash_{wf}$
 $| wf\text{-}Cons: \Gamma \vdash_{wfB}\ B \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow B :: \Gamma \vdash_{wf}$

The judgement $\Gamma \vdash_{wfB}\ B$, which denotes well-formedness of the binding B with respect to context Γ , is just an abbreviation for $\Gamma \vdash_{wf}\ type\text{-}ofB\ B$. 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*:

$\Gamma \vdash_{wf}\ TVar\ i$
 $\Gamma \vdash_{wf}\ Top$
 $\Gamma \vdash_{wf}\ T \rightarrow U$
 $\Gamma \vdash_{wf}\ (\forall <: T. U)$

inductive-cases *well-formedE-cases*:

$B :: \Gamma \vdash_{wf}$

lemma $wf\text{-}TVarB: \Gamma \vdash_{wf}\ T \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow TVarB\ T :: \Gamma \vdash_{wf}$

by (rule $wf\text{-}Cons$) *simp-all*

lemma $wf\text{-}VarB: \Gamma \vdash_{wf}\ T \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow VarB\ T :: \Gamma \vdash_{wf}$

by (rule $wf\text{-}Cons$) *simp-all*

lemma *map-is-TVarb*:

$map\ is\text{-}TVarB\ \Gamma' = map\ is\text{-}TVarB\ \Gamma \Longrightarrow$
 $\Gamma \langle i \rangle = \lfloor TVarB\ T \rfloor \Longrightarrow \exists T. \Gamma' \langle i \rangle = \lfloor TVarB\ T \rfloor$
apply (induct Γ 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 Γ is also well-formed in another context Γ' that contains type variable bindings at the same positions as Γ :

```

lemma wf-equallength:
  assumes  $H: \Gamma \vdash_{wf} T$ 
  shows  $map\ is-TVarB\ \Gamma' = map\ is-TVarB\ \Gamma \implies \Gamma' \vdash_{wf} T$  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:
   $\Delta @ B :: \Gamma \vdash_{wf} \implies \Gamma \vdash_{wfB} B' \implies is-TVarB\ B' = is-TVarB\ B \implies$ 
   $\Delta @ B' :: \Gamma \vdash_{wf}$ 
  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
  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 - || $\Delta$ || = 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 (drule-tac x=TVarB T ::  $\Delta$  in meta-spec)
apply simp
done

```

```

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

```

```

lemma wfE-weaken:  $\Delta @ \Gamma \vdash_{wf} \implies \Gamma \vdash_{wfB} B \implies \uparrow_e (Suc 0) 0 \Delta @ B :: \Gamma$ 
 $\vdash_{wf}$ 
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

```

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 func-

tions. The typing judgement, which will be described in §2.5, involves the lookup of variables in a context. It has already been pointed out earlier that each entry in a context may only depend on types declared “further to the right”. To ensure that a type T stored at position i in an environment Γ is valid in the full environment, as opposed to the smaller environment consisting only of the entries in Γ at positions greater than i , we need to increment the indices of all free type variables in T by $Suc\ i$:

lemma *wf-liftB*:

assumes $H: \Gamma \vdash_{wf}$
shows $\Gamma \langle i \rangle = \lfloor VarB\ T \rfloor \implies \Gamma \vdash_{wf} \uparrow_{\tau} (Suc\ i)\ 0\ T$
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_{τ} (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*:

$\Delta @ B :: \Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} U \implies \Delta[0 \mapsto_{\tau} U]_e @ \Gamma \vdash_{wf} T[\|\Delta\| \mapsto_{\tau} U]_{\tau}$
apply (*induct T arbitrary: Δ*)
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  $\wedge x. PROP P x$  for  $P :: - \Rightarrow prop$ )
apply (drule-tac  $x = TVarB T1 :: \Delta$  in meta-spec)
apply simp
done

```

```

theorem wfE-subst:  $\Delta @ B :: \Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} U \implies \Delta[0 \mapsto_{\tau} U]_e @ \Gamma \vdash_{wf}$ 
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:  $\Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} S \implies \Gamma \vdash S <: Top$ 
| SA-refl-TVar:  $\Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} TVar i \implies \Gamma \vdash TVar i <: TVar i$ 
| SA-trans-TVar:  $\Gamma \langle i \rangle = [TVarB U] \implies$ 
   $\Gamma \vdash \uparrow_{\tau} (Suc i) 0 U <: T \implies \Gamma \vdash TVar i <: T$ 
| SA-arrow:  $\Gamma \vdash T_1 <: S_1 \implies \Gamma \vdash S_2 <: T_2 \implies \Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2$ 
| SA-all:  $\Gamma \vdash T_1 <: S_1 \implies TVarB T_1 :: \Gamma \vdash S_2 <: T_2 \implies$ 
   $\Gamma \vdash (\forall <: S_1. S_2) <: (\forall <: T_1. T_2)$ 

```

The rules *SA-Top* and *SA-refl-TVar*, which appear at the leaves of the derivation tree for a judgement $\Gamma \vdash T <: U$, contain additional side conditions ensuring the well-formedness of the contexts and types involved. In order for the rule *SA-trans-TVar* to be applicable, the context Γ must be

of the form $\Gamma_1 @ B :: \Gamma_2$, where Γ_1 has the length i . Since the indices of variables in B can only refer to variables defined in Γ_2 , they have to be incremented by $Suc\ i$ to ensure that they point to the right variables in the larger context Γ .

lemma *wf-subtype-env*:
assumes $PQ: \Gamma \vdash P <: Q$
shows $\Gamma \vdash_{wf} P$ **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 $H': \Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} U \implies P$
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*:
 $\Gamma \vdash T <: U \implies \Gamma \vdash_{wf} T \wedge \Gamma \vdash_{wf} U$
by (*erule wf-subtypeE*) *iprover*

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

lemma *subtype-refl*: — A.1
 $\Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} T \implies \Gamma \vdash T <: T$
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 $wf: \Gamma \vdash_{wf} B$
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 < ||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 Δ to a context Γ preserves subtyping can be proved by induction on Δ , using the previous result in the induction step:

lemma *subtype-weaken'*: — A.2

$$\Gamma \vdash P <: Q \implies \Delta @ \Gamma \vdash_{wf} \implies \Delta @ \Gamma \vdash \uparrow_{\tau} \|\Delta\| 0 P <: \uparrow_{\tau} \|\Delta\| 0 Q$$

apply (*induct* Δ)

```

apply simp-all
apply (erule well-formedE-cases)
apply simp
apply (drule-tac B=a and  $\Gamma=\Delta @ \Gamma$  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 $\Delta @ TVarB P :: \Gamma \vdash \uparrow_\tau (i + 1) \theta Q <: T$ and $(\Delta @ TVarB Q :: \Gamma)(i) = \lfloor TVarB Q \rfloor$. By assumption, we have $\Gamma \vdash P <: Q$ and hence $\Delta @ TVarB P :: \Gamma \vdash \uparrow_\tau (i + 1) \theta P <: \uparrow_\tau (i + 1) \theta Q$ by weakening. Since $\uparrow_\tau (i + 1) \theta Q$ has the same size as Q , we can use the transitivity property, which yields $\Delta @ TVarB P :: \Gamma \vdash \uparrow_\tau (i + 1) \theta P <: T$. The claim then follows easily by an application of *SA-trans-TVar*.

lemma *subtype-trans*: — A.3

```

 $\Gamma \vdash S <: Q \implies \Gamma \vdash Q <: T \implies \Gamma \vdash S <: T$ 
 $\Delta @ TVarB Q :: \Gamma \vdash M <: N \implies \Gamma \vdash P <: Q \implies$ 
 $\Delta @ TVarB P :: \Gamma \vdash M <: N$ 

```

using *wf-measure-size*

proof (*induct Q arbitrary: $\Gamma S T \Delta P M N$ rule: wf-induct-rule*)

case (*less Q*)

```

{
  fix  $\Gamma S T Q'$ 
  assume  $\Gamma \vdash S <: Q'$ 
  then have  $\Gamma \vdash Q' <: T \implies size Q = size Q' \implies \Gamma \vdash S <: T$ 
  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-reft-TVar show ?case by fact
  next
  case SA-trans-TVar
  thus ?case by (auto intro: subtyping.SA-trans-TVar)

```



```

next
  case (SA-arrow  $\Gamma$   $T_1$   $S_1$   $S_2$   $T_2$ )
  note SA-arrow' = SA-arrow
  from SA-arrow(5) show ?case
  proof cases
    case SA-Top
    with SA-arrow show ?thesis
    by (auto intro: subtyping.SA-Top wf-arrow elim: wf-subtypeE)
  next
  case (SA-arrow  $T_1'$   $T_2'$ )
  from SA-arrow SA-arrow' have  $\Gamma \vdash S_1 \rightarrow S_2 <: T_1' \rightarrow T_2'$ 
  by (auto intro!: subtyping.SA-arrow intro: less(1) [of  $T_1$ ] less(1) [of  $T_2$ ])
  with SA-arrow show ?thesis by simp
qed
next
  case (SA-all  $\Gamma$   $T_1$   $S_1$   $S_2$   $T_2$ )
  note SA-all' = SA-all
  from SA-all(5) show ?case
  proof cases
    case SA-Top
    with SA-all show ?thesis by (auto intro!:
      subtyping.SA-Top wf-all intro: wf-equallength elim: wf-subtypeE)
  next
  case (SA-all  $T_1'$   $T_2'$ )
  from SA-all SA-all' have  $\Gamma \vdash T_1' <: S_1$ 
  by - (rule less(1), simp-all)
  moreover from SA-all SA-all' have  $TVarB\ T_1' :: \Gamma \vdash S_2 <: T_2$ 
  by - (rule less(2) [of - []], simplified), simp-all)
  with SA-all SA-all' have  $TVarB\ T_1' :: \Gamma \vdash S_2 <: T_2'$ 
  by - (rule less(1), simp-all)
  ultimately have  $\Gamma \vdash (\forall <: S_1. S_2) <: (\forall <: T_1'. T_2')$ 
  by (rule subtyping.SA-all)
  with SA-all show ?thesis by simp
qed
qed
}
note tr = this
{
  case 1
  thus ?case using refl by (rule tr)
next
  case 2
  from 2(1) show  $\Delta @ TVarB\ P :: \Gamma \vdash M <: N$ 
  proof (induct  $\Delta @ TVarB\ Q :: \Gamma\ M\ N$  arbitrary:  $\Delta$ )
    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 $H: \Delta @ TVarB Q :: \Gamma \vdash S <: T$

shows $\Gamma \vdash P <: Q \implies \Delta[0 \mapsto_{\tau} P]_e @ \Gamma \vdash S[||\Delta|| \mapsto_{\tau} P]_{\tau} <: T[||\Delta|| \mapsto_{\tau} P]_{\tau}$
using H
apply (*induct* $\Delta @ TVarB Q :: \Gamma S T$ *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  $\|\Delta\| \leq i - \text{Suc } 0$ )
apply (rename-tac nat)
apply (subgoal-tac  $i - \text{Suc } \|\Delta\| = \text{nat}$ )
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  $\text{Suc } (\|\Delta\| - \text{Suc } 0) = \|\Delta\|$ )
apply (simp (no-asm-simp))
apply arith
apply (rule SA-arrow)
apply simp+
apply (rule SA-all)
apply simp
apply (drule-tac  $x = \text{TVarB } T_1 :: \Delta$  in meta-spec)
apply simp
done

```

lemma *subst-subtype*:

```

assumes  $H: \Delta @ \text{VarB } V :: \Gamma \vdash T <: U$ 
shows  $\downarrow_e 1 0 \Delta @ \Gamma \vdash \downarrow_\tau 1 \|\Delta\| T <: \downarrow_\tau 1 \|\Delta\| U$ 
using  $H$ 
apply (induct  $\Delta @ \text{VarB } V :: \Gamma T U$  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  $\|\Delta\| \leq i - \text{Suc } 0$ )
apply (rename-tac nat)
apply (subgoal-tac  $i - \text{Suc } \|\Delta\| = \text{nat}$ )
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 (drule-tac  $x = \text{TVarB } T_1 :: \Delta$  in meta-spec)
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* \Rightarrow *trm* \Rightarrow *type* \Rightarrow *bool* ($- \vdash - : - [50, 50, 50] 50$)

where

$T\text{-Var}$: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \langle i \rangle = \lfloor \text{VarB } U \rfloor \Longrightarrow T = \uparrow_{\tau} (\text{Suc } i) \ 0 \ U \Longrightarrow \Gamma \vdash \text{Var } i : T$
 $T\text{-Abs}$: $\text{VarB } T_1 :: \Gamma \vdash t_2 : T_2 \Longrightarrow \Gamma \vdash (\lambda:T_1. t_2) : T_1 \rightarrow \downarrow_{\tau} 1 \ 0 \ T_2$
 $T\text{-App}$: $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \Longrightarrow \Gamma \vdash t_2 : T_{11} \Longrightarrow \Gamma \vdash t_1 \cdot t_2 : T_{12}$
 $T\text{-TAbs}$: $\text{TVarB } T_1 :: \Gamma \vdash t_2 : T_2 \Longrightarrow \Gamma \vdash (\lambda<:T_1. t_2) : (\forall<:T_1. T_2)$
 $T\text{-TApp}$: $\Gamma \vdash t_1 : (\forall<:T_{11}. T_{12}) \Longrightarrow \Gamma \vdash T_2 <: T_{11} \Longrightarrow$
 $\Gamma \vdash t_1 \cdot_{\tau} T_2 : T_{12}[0 \mapsto_{\tau} T_2]_{\tau}$
 $T\text{-Sub}$: $\Gamma \vdash t : S \Longrightarrow \Gamma \vdash S <: T \Longrightarrow \Gamma \vdash t : T$

Note that in the rule $T\text{-Var}$, the indices of the type U looked up in the context Γ need to be incremented in order for the type to be well-formed with respect to Γ . In the rule $T\text{-Abs}$, 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 $\text{Var}B\ T_1$ 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 $\Gamma \vdash_{wf} \text{using } H$
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*: $\Gamma \vdash t : T \implies \Gamma \vdash_{wf} \wedge \Gamma \vdash_{wf} T$
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  $\Gamma \vdash P <: Q \implies \Delta @ TVarB P :: \Gamma \vdash t : T$ 
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 (drule-tac  $x = VarB T_1 :: \Delta$  in meta-spec)
apply simp
apply (rule-tac  $T_{11} = T_{11}$  in  $T$ -App)
apply simp+
apply (rule  $T$ -TAbs)
apply (drule-tac  $x = TVarB T_1 :: \Delta$  in meta-spec)
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  $\Gamma \vdash_{wf} t$  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  $\Gamma \vdash T <: U \rightarrow U' \implies$ 
  ( $\bigwedge S'. \Gamma \vdash U <: S \implies VarB S :: \Gamma \vdash s : S' \implies$ 
    $\Gamma \vdash \downarrow_{\tau} 1 0 S' <: U' \implies P$ )  $\implies P$ 
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 $\langle \Gamma \vdash T_1 \rightarrow \downarrow_{\tau} 1 0 T_2 <: U \rightarrow U' \rangle$
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** $\langle \Gamma \vdash T <: U \rightarrow U' \rangle$
have $\Gamma \vdash S' <: U \rightarrow U'$ **by** (*rule subtype-trans(1)*)
then show $?case$
by (*rule T-Sub*) (*rule T-Sub(5)*)
qed

lemma *Abs-type'*:
assumes $H: \Gamma \vdash (\lambda:S. s) : U \rightarrow U'$
and $R: \bigwedge S'. \Gamma \vdash U <: S \implies VarB S :: \Gamma \vdash s : S' \implies$
 $\Gamma \vdash \downarrow_{\tau} 1 0 S' <: U' \implies P$
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 $\Gamma \vdash T <: (\forall <:U. U') \implies$
 $(\bigwedge S'. \Gamma \vdash U <: S \implies TVarB U :: \Gamma \vdash s : S' \implies$
 $TVarB U :: \Gamma \vdash S' <: U' \implies P) \implies P$
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 $\langle \Gamma \vdash (\forall <:T_1. T_2) <: (\forall <:U. U') \rangle$
obtain $ty1: \Gamma \vdash U <: T_1$ **and** $ty2: TVarB U :: \Gamma \vdash T_2 <: U'$
by *cases simp-all*
from $\langle TVarB T_1 :: \Gamma \vdash t_2 : T_2 \rangle$
have $TVarB U :: \Gamma \vdash t_2 : T_2$ **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 $\langle \Gamma \vdash S' <: T \rangle$ **and** $\langle \Gamma \vdash T <: (\forall <:U. U') \rangle$
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 $R: \bigwedge S'. \Gamma \vdash U <: S \implies TVarB U :: \Gamma \vdash s : S' \implies$
 $TVarB U :: \Gamma \vdash S' <: U' \implies P$

shows P **using** H *subtype-refl'* [$OF\ H$]
by (*rule TAbs-type*) (*rule R*)

lemma $T\text{-eq}$: $\Gamma \vdash t : T \implies T = T' \implies \Gamma \vdash t : T'$ **by** *simp*

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

lemma *type-weaken*:
assumes H : $\Delta @ \Gamma \vdash t : T$
shows $\Gamma \vdash_{wfB} B \implies$
 $\uparrow_e\ 1\ 0\ \Delta @ B :: \Gamma \vdash \uparrow\ 1\ \|\Delta\| t : \uparrow_\tau\ 1\ \|\Delta\| T$ **using** H
apply (*induct* $\Delta @ \Gamma t T$ *arbitrary*: Δ)
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 (*drule-tac* $x = VarB\ T_1 :: \Delta$ *in meta-spec*)
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 (*drule-tac* $x = TVarB\ T_1 :: \Delta$ *in meta-spec*)
apply *simp*
apply (*erule-tac T-TApp* [*THEN T-eq*])
apply (*drule subtype-weaken*)
apply *simp+*
apply (*case-tac* Δ)
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 Δ to the context Γ 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* Δ)
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 Δ , using the previous result in the induction step. In the proof of the preservation theorem, we will need two substitution theorems for term and type variables, both of which are proved by induction on the typing derivation. Since term and type variables are stored in the same context, we again have to decrement the free type variables in Δ and T by 1 in the substitution rule for term variables in order to compensate for the disappearance of the variable.

theorem *subst-type*: — A.8
assumes $H: \Delta @ \text{Var}B \ U :: \Gamma \vdash t : T$
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* $\Delta @ \text{Var}B \ U :: \Gamma \ t \ T$ *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 ( $\|\Delta\| - \text{Suc } 0 = \|\Delta\|$ ))
apply (simp (no-asm-simp))
apply arith
apply simp
apply (drule-tac  $x = \text{VarB } T_1 :: \Delta$  in meta-spec)
apply (rule T-Abs [THEN T-eq])
apply simp
apply (simp add: substT-substT [symmetric])
apply simp
apply (rule-tac  $T_{11} = T_{11}[\|\Delta\| \mapsto_\tau \text{Top}]_\tau$  in T-App)
apply simp+
apply (rule T-TAbs)
apply (drule-tac  $x = \text{TVarB } T_1 :: \Delta$  in meta-spec)
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  $S = S[\|\Delta\| \mapsto_\tau \text{Top}]_\tau$  in T-Sub)
apply simp
apply simp
apply (rule subst-subtype [simplified])
apply assumption
done

```

theorem *substT-type*: — A.11

```

assumes  $H: \Delta @ \text{TVarB } Q :: \Gamma \vdash t : T$ 
shows  $\Gamma \vdash P <: Q \implies$ 
   $\Delta[0 \mapsto_\tau P]_e @ \Gamma \vdash t[\|\Delta\| \mapsto_\tau P] : T[\|\Delta\| \mapsto_\tau P]_\tau$  using  $H$ 
apply (induct  $\Delta @ \text{TVarB } Q :: \Gamma \vdash t : T$  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  $\|\Delta\| \leq i - \text{Suc } 0$ )
apply (rename-tac nat)
apply (subgoal-tac  $i - \text{Suc } \|\Delta\| = \text{nat}$ )
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 ( $\|\Delta\| - \text{Suc } 0$ ) =  $\|\Delta\|$ )
apply (simp (no-asm-simp))
apply arith
apply arith
apply (rule T-Abs [THEN T-eq])
apply (drule-tac x=VarB T1 ::  $\Delta$  in meta-spec)
apply simp
apply (simp add: substT-substT [symmetric])
apply (rule-tac T11=T11[ $\|\Delta\| \mapsto_\tau P$ ] $\tau$  in T-App)
apply simp+
apply (rule T-TAbs)
apply (drule-tac x=TVarB T1 ::  $\Delta$  in meta-spec)
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: $(\lambda:T. t) \in \textit{value}$

| *TAbs*: $(\lambda<:T. t) \in \textit{value}$

The notion of a *value* is now used in the definition of the evaluation relation $t \mapsto t'$. There are several ways for defining this evaluation relation: Ay-

demir et al. [1] advocate the use of *evaluation contexts* that allow to separate the description of the “immediate” reduction rules, i.e. β -reduction, from the description of the context in which these reductions may occur in. The rationale behind this approach is to keep the formalization more modular. We will take a closer look at this style of presentation in section §4. For the rest of this section, we will use a different approach: both the “immediate” reductions and the reduction context are described within the same inductive definition, where the context is described by additional congruence rules.

inductive

$eval :: trm \Rightarrow trm \Rightarrow bool$ (**infixl** \mapsto 50)

where

$E\text{-Abs}: v_2 \in value \Longrightarrow (\lambda:T_{11}. t_{12}) \cdot v_2 \mapsto t_{12}[0 \mapsto v_2]$
 $| E\text{-TAbs}: (\lambda<:T_{11}. t_{12}) \cdot_{\tau} T_2 \mapsto t_{12}[0 \mapsto_{\tau} T_2]$
 $| E\text{-App1}: t \mapsto t' \Longrightarrow t \cdot u \mapsto t' \cdot u$
 $| E\text{-App2}: v \in value \Longrightarrow t \mapsto t' \Longrightarrow v \cdot t \mapsto v \cdot t'$
 $| E\text{-TApp}: t \mapsto t' \Longrightarrow t \cdot_{\tau} T \mapsto t' \cdot_{\tau} T$

Here, the rules $E\text{-Abs}$ and $E\text{-TAbs}$ describe the “immediate” reductions, whereas $E\text{-App1}$, $E\text{-App2}$, and $E\text{-TApp}$ 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 $t \mapsto t'$.

theorem *preservation*: — A.20

assumes $H: \Gamma \vdash t : T$

shows $t \mapsto t' \Longrightarrow \Gamma \vdash t' : T$ **using** H

proof (*induct arbitrary: t'*)

case ($T\text{-Var } \Gamma \ i \ U \ T \ t'$)

from $\langle Var \ i \ \mapsto \ t' \rangle$

show *?case by cases*

next

case ($T\text{-Abs } T_1 \ \Gamma \ t_2 \ T_2 \ t'$)

from $\langle (\lambda:T_1. t_2) \ \mapsto \ t' \rangle$

show *?case by cases*

next

case ($T\text{-App } \Gamma \ t_1 \ T_{11} \ T_{12} \ t_2 \ t'$)

from $\langle t_1 \cdot t_2 \ \mapsto \ t' \rangle$

show *?case*

proof *cases*

case ($E\text{-Abs } T_{11}' \ t_{12}$)

with $T\text{-App}$ **have** $\Gamma \vdash (\lambda:T_{11}'. t_{12}) : T_{11} \rightarrow T_{12}$ **by** *simp*

then obtain S'

where $T_{11}: \Gamma \vdash T_{11} <: T_{11}'$

```

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

```

next
 case $(T\text{-Sub } \Gamma \ t \ S \ T \ t')$
 from $\langle t \mapsto t' \rangle$
 have $\Gamma \vdash t' : S$ by (rule $T\text{-Sub}$)
 then show $?case$ using $\langle \Gamma \vdash S <: T \rangle$
 by (rule *typing.T-Sub*)
qed

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

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

lemma *TyAll-canonical*: — A.14(3)
 assumes $ty: \square \vdash v : (\forall <: T_1. T_2)$
 shows $v \in \text{value} \implies \exists t \ S. v = (\lambda <: S. t)$ using ty
proof (induct $\square :: env \ v \ \forall <: T_1. T_2$ arbitrary: $T_1 \ T_2$)
 case $(T\text{-App } t_1 \ T_{11} \ t_2 \ T_1 \ T_2)$
 from $\langle t_1 \cdot t_2 \in \text{value} \rangle$
 show $?case$ by *cases*
next
 case $T\text{-TAbs}$
 show $?case$ by *iprover*
next
 case $(T\text{-TApp } t_1 \ T_{11} \ T_{12} \ T_2 \ T_1 \ T_2')$
 from $\langle t_1 \cdot_{\tau} T_2 \in \text{value} \rangle$
 show $?case$ by *cases*
next

```

case ( $T\text{-Sub } t S T_1 T_2$ )
from  $\langle \square \vdash S <: (\forall <: T_1. T_2) \rangle$ 
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  $ty: \square \vdash t : T$ 
  shows  $t \in \text{value} \vee (\exists t'. t \mapsto t')$  using  $ty$ 
proof (induct []::env t T)
  case  $T\text{-Var}$ 
  thus  $?case$  by simp
next
  case  $T\text{-Abs}$ 
  from  $\text{value.Abs}$  show  $?case ..$ 
next
  case ( $T\text{-App } t_1 T_{11} T_{12} t_2$ )
  hence  $t_1 \in \text{value} \vee (\exists t'. t_1 \mapsto t')$  by simp
  thus  $?case$ 
  proof
    assume  $t_1\text{-val}: t_1 \in \text{value}$ 
    with  $T\text{-App}$  obtain  $t S$  where  $t_1: t_1 = (\lambda:S. t)$ 
    by (auto dest!: Fun-canonical)
    from  $T\text{-App}$  have  $t_2 \in \text{value} \vee (\exists t'. t_2 \mapsto t')$  by simp
    thus  $?thesis$ 
    proof
      assume  $t_2 \in \text{value}$ 
      with  $t_1$  have  $t_1 \cdot t_2 \mapsto t[0 \mapsto t_2]$ 
      by simp (rule eval.intros)
      thus  $?thesis$  by iprover
    next
      assume  $\exists t'. t_2 \mapsto t'$ 
      then obtain  $t'$  where  $t_2 \mapsto t'$  by iprover
      with  $t_1\text{-val}$  have  $t_1 \cdot t_2 \mapsto t_1 \cdot t'$  by (rule eval.intros)
      thus  $?thesis$  by iprover
    qed
  next
    assume  $\exists t'. t_1 \mapsto t'$ 
    then obtain  $t'$  where  $t_1 \mapsto t' ..$ 
    hence  $t_1 \cdot t_2 \mapsto t' \cdot t_2$  by (rule eval.intros)
    thus  $?thesis$  by iprover
  qed
next
  case  $T\text{-TAbs}$ 
  from  $\text{value.TAbs}$  show  $?case ..$ 
next
  case ( $T\text{-TApp } t_1 T_{11} T_{12} T_2$ )
  hence  $t_1 \in \text{value} \vee (\exists t'. t_1 \mapsto t')$  by simp

```



```

thus ?case
proof
  assume  $t_1 \in \text{value}$ 
  with  $T\text{-TApp}$  obtain  $t S$  where  $t_1 = (\lambda <: S. t)$ 
  by (auto dest! TyAll-canonical)
  hence  $t_1 \cdot_\tau T_2 \mapsto t[0 \mapsto_\tau T_2]$  by (simp (rule eval.intros))
  thus ?thesis by iprover
next
  assume  $\exists t'. t_1 \mapsto t'$ 
  then obtain  $t'$  where  $t_1 \mapsto t' ..$ 
  hence  $t_1 \cdot_\tau T_2 \mapsto t' \cdot_\tau T_2$  by (rule eval.intros)
  thus ?thesis by iprover
qed
next
  case ( $T\text{-Sub } t S T$ )
  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 $\text{name} = \text{string}$

```

datatype  $\text{type} =$ 
   $TVar$   $\text{nat}$ 
  |  $Top$ 
  |  $Fun$   $\text{type } \text{type}$  (infixr  $\rightarrow 200$ )
  |  $TyAll$   $\text{type } \text{type}$  ( $(\exists \forall <: \cdot / \cdot) [0, 10] 10$ )
  |  $RcdT$  ( $\text{name} \times \text{type}$ )  $\text{list}$ 

```

type-synonym $\text{fldT} = \text{name} \times \text{type}$

type-synonym $\text{rcdT} = (\text{name} \times \text{type}) \text{list}$

datatype $\text{binding} = VarB \text{type} \mid TVarB \text{type}$

type-synonym $\text{env} = \text{binding list}$

primrec $\text{is-TVarB} :: \text{binding} \Rightarrow \text{bool}$

where

$is-TV\text{ar}B (VarB T) = False$
| $is-TV\text{ar}B (TVarB T) = True$

primrec $type\text{-of}B :: binding \Rightarrow type$

where

$type\text{-of}B (VarB T) = T$
| $type\text{-of}B (TVarB T) = T$

primrec $mapB :: (type \Rightarrow type) \Rightarrow binding \Rightarrow binding$

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 $pat = PVar\ type \mid PRcd\ (name \times pat)\ list$

datatype $trm =$

$Var\ nat$
| $Abs\ type\ trm\ ((\exists\lambda:-./ -)\ [0, 10]\ 10)$
| $TAbs\ type\ trm\ ((\exists\lambda<:-./ -)\ [0, 10]\ 10)$
| $App\ trm\ trm\ (\mathbf{infixl}\ \cdot\ 200)$
| $TApp\ trm\ type\ (\mathbf{infixl}\ \cdot_{\tau}\ 200)$
| $Rcd\ (name \times trm)\ list$
| $Proj\ trm\ name\ ((-./ -)\ [90, 91]\ 90)$
| $LET\ pat\ trm\ trm\ ((LET\ (- = / -)\ IN\ (-))\ 10)$

type-synonym $fld = name \times trm$

type-synonym $rcd = (name \times trm)\ list$

type-synonym $fpat = name \times pat$

type-synonym $rpat = (name \times pat)\ list$

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

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

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

3.2 Lifting and Substitution

primrec $psize :: pat \Rightarrow nat (\|- \|_p)$
and $rsize :: rpat \Rightarrow nat (\|- \|_r)$
and $fsize :: fpat \Rightarrow nat (\|- \|_f)$

where

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

primrec $liftT :: nat \Rightarrow nat \Rightarrow type \Rightarrow type (\uparrow_\tau)$
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

$\uparrow_\tau\ n\ k\ (TVar\ i) = (if\ i < k\ then\ TVar\ i\ else\ TVar\ (i + n))$
 $\uparrow_\tau\ n\ k\ Top = Top$
 $\uparrow_\tau\ n\ k\ (T \rightarrow U) = \uparrow_\tau\ n\ k\ T \rightarrow \uparrow_\tau\ n\ k\ U$
 $\uparrow_\tau\ n\ k\ (\forall <: T. U) = (\forall <: \uparrow_\tau\ n\ k\ T. \uparrow_\tau\ n\ (k + 1)\ U)$
 $\uparrow_\tau\ n\ k\ (RcdT\ fs) = RcdT\ (\uparrow_{r\tau}\ n\ k\ fs)$
 $\uparrow_{r\tau}\ n\ k\ \square = \square$
 $\uparrow_{r\tau}\ n\ k\ (f :: fs) = \uparrow_{f\tau}\ n\ k\ f :: \uparrow_{r\tau}\ n\ k\ fs$
 $\uparrow_{f\tau}\ n\ k\ (l, T) = (l, \uparrow_\tau\ n\ k\ T)$

primrec $liftp :: nat \Rightarrow nat \Rightarrow pat \Rightarrow pat (\uparrow_p)$
and $liftfp :: nat \Rightarrow nat \Rightarrow rpat \Rightarrow rpat (\uparrow_{rp})$
and $liftfp :: nat \Rightarrow nat \Rightarrow fpat \Rightarrow fpat (\uparrow_{fp})$

where

$\uparrow_p\ n\ k\ (PVar\ T) = PVar\ (\uparrow_\tau\ n\ k\ T)$
 $\uparrow_p\ n\ k\ (PRcd\ fs) = PRcd\ (\uparrow_{rp}\ n\ k\ fs)$
 $\uparrow_{rp}\ n\ k\ \square = \square$
 $\uparrow_{rp}\ n\ k\ (f :: fs) = \uparrow_{fp}\ n\ k\ f :: \uparrow_{rp}\ n\ k\ fs$
 $\uparrow_{fp}\ n\ k\ (l, p) = (l, \uparrow_p\ n\ k\ p)$

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

$\uparrow\ n\ k\ (Var\ i) = (if\ i < k\ then\ Var\ i\ else\ Var\ (i + n))$
 $\uparrow\ n\ k\ (\lambda:T. t) = (\lambda:\uparrow_\tau\ n\ k\ T. \uparrow\ n\ (k + 1)\ t)$
 $\uparrow\ n\ k\ (\lambda<:T. t) = (\lambda<:\uparrow_\tau\ n\ k\ T. \uparrow\ n\ (k + 1)\ t)$
 $\uparrow\ n\ k\ (s \cdot t) = \uparrow\ n\ k\ s \cdot \uparrow\ n\ k\ t$
 $\uparrow\ n\ k\ (t \cdot_\tau\ T) = \uparrow\ n\ k\ t \cdot_\tau\ \uparrow_\tau\ n\ k\ T$
 $\uparrow\ n\ k\ (Rcd\ fs) = Rcd\ (\uparrow_r\ n\ k\ fs)$
 $\uparrow\ n\ k\ (t..a) = (\uparrow\ n\ k\ t)..a$
 $\uparrow\ n\ k\ (LET\ p = t\ IN\ u) = (LET\ \uparrow_p\ n\ k\ p = \uparrow\ n\ k\ t\ IN\ \uparrow\ n\ (k + \|p\|_p)\ u)$
 $\uparrow_r\ n\ k\ \square = \square$
 $\uparrow_r\ n\ k\ (f :: fs) = \uparrow_f\ n\ k\ f :: \uparrow_r\ n\ k\ fs$
 $\uparrow_f\ n\ k\ (l, t) = (l, \uparrow\ n\ k\ t)$

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

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

primrec $substP :: pat \Rightarrow nat \Rightarrow type \Rightarrow pat \ (-[- \mapsto_{\tau} -]_p [300, 0, 0] 300)$
and $substrpT :: rpat \Rightarrow nat \Rightarrow type \Rightarrow rpat \ (-[- \mapsto_{\tau} -]_{rp} [300, 0, 0] 300)$
and $substfpT :: fpat \Rightarrow nat \Rightarrow type \Rightarrow fpat \ (-[- \mapsto_{\tau} -]_{fp} [300, 0, 0] 300)$
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 $decp :: nat \Rightarrow nat \Rightarrow pat \Rightarrow pat \ (\downarrow_p)$

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 $subst :: trm \Rightarrow nat \Rightarrow trm \Rightarrow trm \ (-[- \mapsto -] [300, 0, 0] 300)$
and $substr :: rcd \Rightarrow nat \Rightarrow trm \Rightarrow rcd \ (-[- \mapsto -]_r [300, 0, 0] 300)$
and $substf :: fld \Rightarrow nat \Rightarrow trm \Rightarrow fld \ (-[- \mapsto -]_f [300, 0, 0] 300)$

where

$(Var\ i)[k \mapsto s] =$
(if $k < i$ *then* $Var\ (i - 1)$ *else if* $i = k$ *then* $\uparrow\ k\ 0\ s$ *else* $Var\ i)$
 $| (t \cdot u)[k \mapsto s] = t[k \mapsto s] \cdot u[k \mapsto s]$
 $| (t \cdot_{\tau} T)[k \mapsto s] = t[k \mapsto s] \cdot_{\tau} T[k \mapsto_{\tau} Top]_{\tau}$
 $| (\lambda:T. t)[k \mapsto s] = (\lambda:T[k \mapsto_{\tau} Top]_{\tau}. t[k+1 \mapsto s])$
 $| (\lambda<:T. t)[k \mapsto s] = (\lambda<:T[k \mapsto_{\tau} Top]_{\tau}. t[k+1 \mapsto s])$
 $| (Rcd\ fs)[k \mapsto s] = Rcd\ (fs[k \mapsto s]_r)$
 $| (t..a)[k \mapsto s] = (t[k \mapsto s])..a$
 $| (LET\ p = t\ IN\ u)[k \mapsto s] = (LET\ \downarrow_p\ 1\ k\ p = t[k \mapsto s]\ IN\ u[k + \|p\|_p \mapsto s])$
 $| [][k \mapsto s]_r = []$

| $(f :: fs)[k \mapsto s]_r = f[k \mapsto s]_f :: fs[k \mapsto s]_r$
| $(l, t)[k \mapsto s]_f = (l, t[k \mapsto s])$

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

primrec *substT* :: *trm* \Rightarrow *nat* \Rightarrow *type* \Rightarrow *trm* $([- \mapsto_\tau -] [300, 0, 0] 300)$
and *substR* :: *rcd* \Rightarrow *nat* \Rightarrow *type* \Rightarrow *rcd* $([- \mapsto_\tau -]_r [300, 0, 0] 300)$
and *substF* :: *fld* \Rightarrow *nat* \Rightarrow *type* \Rightarrow *fld* $([- \mapsto_\tau -]_f [300, 0, 0] 300)$
where

$(Var\ i)[k \mapsto_\tau S] = (if\ k < i\ then\ Var\ (i - 1)\ else\ Var\ i)$
| $(t \cdot u)[k \mapsto_\tau S] = t[k \mapsto_\tau S] \cdot u[k \mapsto_\tau S]$
| $(t \cdot_\tau T)[k \mapsto_\tau S] = t[k \mapsto_\tau S] \cdot_\tau T[k \mapsto_\tau S]_\tau$
| $(\lambda:T. t)[k \mapsto_\tau S] = (\lambda:T[k \mapsto_\tau S]_\tau. t[k+1 \mapsto_\tau S])$
| $(\lambda<:T. t)[k \mapsto_\tau S] = (\lambda<:T[k \mapsto_\tau S]_\tau. t[k+1 \mapsto_\tau S])$
| $(Rcd\ fs)[k \mapsto_\tau S] = Rcd\ (fs[k \mapsto_\tau S]_r)$
| $(t..a)[k \mapsto_\tau S] = (t[k \mapsto_\tau S])..a$
| $(LET\ p = t\ IN\ u)[k \mapsto_\tau S] =$
 $(LET\ p[k \mapsto_\tau S]_p = t[k \mapsto_\tau S]\ IN\ u[k + \|p\|_p \mapsto_\tau S])$
| $\llbracket k \mapsto_\tau S \rrbracket_r = \llbracket \rrbracket$
| $(f :: fs)[k \mapsto_\tau S]_r = f[k \mapsto_\tau S]_f :: fs[k \mapsto_\tau S]_r$
| $(l, t)[k \mapsto_\tau S]_f = (l, t[k \mapsto_\tau S])$

primrec *liftE* :: *nat* \Rightarrow *nat* \Rightarrow *env* \Rightarrow *env* (\uparrow_e)
where

$\uparrow_e\ n\ k\ \llbracket \rrbracket = \llbracket \rrbracket$
| $\uparrow_e\ n\ k\ (B :: \Gamma) = mapB\ (\uparrow_\tau\ n\ (k + \|\Gamma\|))\ B :: \uparrow_e\ n\ k\ \Gamma$

primrec *substE* :: *env* \Rightarrow *nat* \Rightarrow *type* \Rightarrow *env* $([- \mapsto_\tau -]_e [300, 0, 0] 300)$
where

$\llbracket k \mapsto_\tau T \rrbracket_e = \llbracket \rrbracket$
| $(B :: \Gamma)[k \mapsto_\tau T]_e = mapB\ (\lambda U. U[k + \|\Gamma\| \mapsto_\tau T]_\tau)\ B :: \Gamma[k \mapsto_\tau T]_e$

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

primrec *substS* :: *trm* \Rightarrow *nat* \Rightarrow *trm list* \Rightarrow *trm* $([- \mapsto_s -] [300, 0, 0] 300)$
where

$t[k \mapsto_s \llbracket \rrbracket] = t$
| $t[k \mapsto_s u :: us] = t[k + \|us\| \mapsto u][k \mapsto_s us]$

primrec *decT* :: *nat* \Rightarrow *nat* \Rightarrow *type* \Rightarrow *type* (\downarrow_τ)
where

$\downarrow_\tau\ 0\ k\ T = T$
| $\downarrow_\tau\ (Suc\ n)\ k\ T = \downarrow_\tau\ n\ k\ (T[k \mapsto_\tau Top]_\tau)$

primrec $decE :: nat \Rightarrow nat \Rightarrow env \Rightarrow env \ (\downarrow_e)$

where

$\downarrow_e 0 k \Gamma = \Gamma$
 $|\ \downarrow_e (Suc\ n) k \Gamma = \downarrow_e\ n\ k\ (\Gamma[k \mapsto_\tau Top]_e)$

primrec $decrT :: nat \Rightarrow nat \Rightarrow rcdT \Rightarrow rcdT \ (\downarrow_{r\tau})$

where

$\downarrow_{r\tau} 0 k fTs = fTs$
 $|\ \downarrow_{r\tau} (Suc\ n) k fTs = \downarrow_{r\tau}\ n\ k\ (fTs[k \mapsto_\tau Top]_{r\tau})$

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

lemma $liftE-length$ [simp]: $\|\uparrow_e\ n\ k\ \Gamma\| = \|\Gamma\|$

by (induct Γ) simp-all

lemma $substE-length$ [simp]: $\|\Gamma[k \mapsto_\tau U]_e\| = \|\Gamma\|$

by (induct Γ) simp-all

lemma $liftE-nth$ [simp]:

$(\uparrow_e\ n\ k\ \Gamma)\langle i \rangle = map-option\ (mapB\ (\uparrow_\tau\ n\ (k + \|\Gamma\| - i - 1)))\ (\Gamma\langle i \rangle)$

apply (induct Γ arbitrary: i)

apply simp

apply simp

apply (case-tac i)

apply simp

apply simp

done

lemma $substE-nth$ [simp]:

$(\Gamma[0 \mapsto_\tau T]_e)\langle i \rangle = map-option\ (mapB\ (\lambda U. U[\|\Gamma\| - i - 1 \mapsto_\tau T]_\tau))\ (\Gamma\langle i \rangle)$

apply (induct Γ arbitrary: i)

apply simp

apply simp

apply (case-tac i)

apply simp

apply simp

done

lemma $liftT-liftT$ [simp]:

$i \leq j \Longrightarrow j \leq i + m \Longrightarrow \uparrow_\tau\ n\ j\ (\uparrow_\tau\ m\ i\ T) = \uparrow_\tau\ (m + n)\ i\ T$

$i \leq j \Longrightarrow j \leq i + m \Longrightarrow \uparrow_{r\tau}\ n\ j\ (\uparrow_{r\tau}\ m\ i\ rT) = \uparrow_{r\tau}\ (m + n)\ i\ rT$

$i \leq j \Longrightarrow j \leq i + m \Longrightarrow \uparrow_{f\tau}\ n\ j\ (\uparrow_{f\tau}\ m\ i\ fT) = \uparrow_{f\tau}\ (m + n)\ i\ fT$

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]:

$i + m \leq j \Longrightarrow \uparrow_\tau\ n\ j\ (\uparrow_\tau\ m\ i\ T) = \uparrow_\tau\ m\ i\ (\uparrow_\tau\ n\ (j - m)\ T)$

$i + m \leq j \implies \uparrow_{r\tau} n j (\uparrow_{r\tau} m i rT) = \uparrow_{r\tau} m i (\uparrow_{r\tau} n (j - m) rT)$
 $i + m \leq j \implies \uparrow_{f\tau} n j (\uparrow_{f\tau} m i fT) = \uparrow_{f\tau} m i (\uparrow_{f\tau} n (j - m) fT)$
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]*:

$k \leq k' \implies k' < k + n \implies (\uparrow_{\tau} n k T)[k' \mapsto_{\tau} U]_{\tau} = \uparrow_{\tau} (n - 1) k T$
 $k \leq k' \implies k' < k + n \implies (\uparrow_{r\tau} n k rT)[k' \mapsto_{\tau} U]_{r\tau} = \uparrow_{r\tau} (n - 1) k rT$
 $k \leq k' \implies k' < k + n \implies (\uparrow_{f\tau} n k fT)[k' \mapsto_{\tau} U]_{f\tau} = \uparrow_{f\tau} (n - 1) k fT$
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]*:

$k \leq k' \implies \uparrow_{\tau} n k (T[k' \mapsto_{\tau} U]_{\tau}) = \uparrow_{\tau} n k T[k' + n \mapsto_{\tau} U]_{\tau}$

$k \leq k' \implies \uparrow_{r\tau} n k (rT[k' \mapsto_\tau U]_{r\tau}) = \uparrow_{r\tau} n k rT[k' + n \mapsto_\tau U]_{r\tau}$
 $k \leq k' \implies \uparrow_{f\tau} n k (fT[k' \mapsto_\tau U]_{f\tau}) = \uparrow_{f\tau} n k fT[k' + n \mapsto_\tau U]_{f\tau}$
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]:

$k' < k \implies$
 $\uparrow_\tau n k (T[k' \mapsto_\tau U]_\tau) = \uparrow_\tau n (k + 1) T[k' \mapsto_\tau \uparrow_\tau n (k - k') U]_\tau$
 $k' < k \implies$
 $\uparrow_{r\tau} n k (rT[k' \mapsto_\tau U]_{r\tau}) = \uparrow_{r\tau} n (k + 1) rT[k' \mapsto_\tau \uparrow_\tau n (k - k') U]_{r\tau}$
 $k' < k \implies$
 $\uparrow_{f\tau} n k (fT[k' \mapsto_\tau U]_{f\tau}) = \uparrow_{f\tau} n (k + 1) fT[k' \mapsto_\tau \uparrow_\tau n (k - k') U]_{f\tau}$
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]:

$k \leq k' \implies \uparrow_\tau n k' (T[k \mapsto_\tau Top]_\tau) = \uparrow_\tau n (Suc k') T[k \mapsto_\tau Top]_\tau$
 $k \leq k' \implies \uparrow_{r\tau} n k' (rT[k \mapsto_\tau Top]_{r\tau}) = \uparrow_{r\tau} n (Suc k') rT[k \mapsto_\tau Top]_{r\tau}$
 $k \leq k' \implies \uparrow_{f\tau} n k' (fT[k \mapsto_\tau Top]_{f\tau}) = \uparrow_{f\tau} n (Suc k') fT[k \mapsto_\tau Top]_{f\tau}$
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]:

$k \leq k' \implies \uparrow_e n k (\Gamma[k' \mapsto_\tau U]_e) = \uparrow_e n k \Gamma[k' + n \mapsto_\tau U]_e$
apply (*induct Γ 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]:

$k \leq k' \implies \uparrow_\tau n k' (\downarrow_\tau m k T) = \downarrow_\tau m k (\uparrow_\tau n (m + k') T)$
by (*induct m arbitrary: T*) *simp-all*

lemma liftT-substT-strange:

$\uparrow_\tau n k T[n + k \mapsto_\tau U]_\tau = \uparrow_\tau n (Suc k) T[k \mapsto_\tau \uparrow_\tau n 0 U]_\tau$
 $\uparrow_{r\tau} n k rT[n + k \mapsto_\tau U]_{r\tau} = \uparrow_{r\tau} n (Suc k) rT[k \mapsto_\tau \uparrow_\tau n 0 U]_{r\tau}$
 $\uparrow_{f\tau} n k fT[n + k \mapsto_\tau U]_{f\tau} = \uparrow_{f\tau} n (Suc k) fT[k \mapsto_\tau \uparrow_\tau n 0 U]_{f\tau}$
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* $\wedge x. PROP P x$ **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 liftp.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 liftp.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* Γ *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* Γ *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 $\wedge x. PROP P x$ 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]:*
 $k \leq j \implies (\downarrow_{\tau} i k T)[j \mapsto_{\tau} U]_{\tau} = \downarrow_{\tau} i k (T[i + j \mapsto_{\tau} U]_{\tau})$
by (*induct i arbitrary: T j*) (*simp-all add: substT-substT [symmetric]*)

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

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

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

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

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

lemma *substE-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 Γ*)
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]:

$k \leq k' \implies k' + m \leq k + n \implies \downarrow_\tau m k' (\uparrow_\tau n k \Gamma) = \uparrow_\tau (n - m) k \Gamma$
apply (induct m arbitrary: n)
apply (subgoal-tac [2] $k' + m \leq k + (n - \text{Suc } 0)$)
apply simp-all
done

theorem *decE-liftE* [simp]:

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

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

apply (induct Γ)
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 Γ) simp-all

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

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

lemma *liftrT-assoc-Some*: $fs\langle l \rangle? = [T] \implies \uparrow_{r\tau} n k fs\langle l \rangle? = [\uparrow_\tau n k T]$
by (induct fs rule: list.induct) auto

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

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

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

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

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

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

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

lemma *substr-assoc-None* [simp]: $(fs[k \mapsto u]_r\langle l \rangle? = \perp) = (fs\langle l \rangle? = \perp)$
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*: $(a, T) \in set fs \implies (a, \uparrow_\tau n k T) \in set (\uparrow_{r\tau} n k fs)$
by (induct fs rule: list.induct) auto

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

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

lemma *substrT-setD*:
 $(a, T) \in set (fs[k \mapsto_\tau U]_{r\tau}) \implies \exists T'. (a, T') \in set fs \wedge T = T'[k \mapsto_\tau U]_\tau$
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 :: $env \Rightarrow type \Rightarrow bool$ (- \vdash_{wf} - [50, 50] 50)

where

wf-TVar: $\Gamma\langle i \rangle = \lfloor TVarB T \rfloor \implies \Gamma \vdash_{wf} TVar i$
| *wf-Top*: $\Gamma \vdash_{wf} Top$
| *wf-arrow*: $\Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} U \implies \Gamma \vdash_{wf} T \rightarrow U$
| *wf-all*: $\Gamma \vdash_{wf} T \implies TVarB T :: \Gamma \vdash_{wf} U \implies \Gamma \vdash_{wf} (\forall \langle : T. U)$
| *wf-RcdT*: $unique fs \implies \forall (l, T) \in set fs. \Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} RcdT fs$

inductive

$well\text{-formed}E :: env \Rightarrow bool \ (- \vdash_{wf} [50] 50)$
and $well\text{-formed}B :: env \Rightarrow binding \Rightarrow bool \ (- \vdash_{wf} B - [50, 50] 50)$
where
 $\Gamma \vdash_{wf} B \equiv \Gamma \vdash_{wf} type\text{-of}B B$
 $| wf\text{-Nil}: [] \vdash_{wf}$
 $| wf\text{-Cons}: \Gamma \vdash_{wf} B \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow B :: \Gamma \vdash_{wf}$

inductive-cases $well\text{-formed-cases}$:

$\Gamma \vdash_{wf} TVar\ i$
 $\Gamma \vdash_{wf} Top$
 $\Gamma \vdash_{wf} T \rightarrow U$
 $\Gamma \vdash_{wf} (\forall <: T. U)$
 $\Gamma \vdash_{wf} (RcdT\ fTs)$

inductive-cases $well\text{-formed}E\text{-cases}$:

$B :: \Gamma \vdash_{wf}$

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

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

lemma $map\text{-is-TVar}b$:

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

lemma $wf\text{-equallength}$:

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

lemma $wfE\text{-replace}$:

$\Delta @ B :: \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} B' \Longrightarrow is\text{-TVar}B\ B' = is\text{-TVar}B\ B \Longrightarrow$
 $\Delta @ B' :: \Gamma \vdash_{wf}$
apply $(induct\ \Delta)$
apply $simp$
apply $(erule\ wf\text{-Cons})$
apply $(erule\ well\text{-formed}E\text{-cases})$
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 (drule-tac  $x = TVar\ B\ T :: \Delta$  in meta-spec)
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'*: $\Gamma \vdash_{wf} T \implies \Delta @ \Gamma \vdash_{wf} \uparrow_\tau \|\Delta\|\ 0\ T$

apply (*induct* Δ)
apply *simp-all*
apply (*drule-tac* $B=a$ **in** *wf-weaken* [*of* [], *simplified*])
apply *simp*
done

lemma *wfE-weaken*: $\Delta @ \Gamma \vdash_{wf} B \implies \Gamma \vdash_{wf} B \implies \uparrow_e (Suc\ 0)\ 0\ \Delta @ B :: \Gamma$

\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

lemma *wf-liftB*:

assumes $H: \Gamma \vdash_{wf}$
shows $\Gamma \langle i \rangle = [VarB\ T] \implies \Gamma \vdash_{wf} \uparrow_\tau (Suc\ i)\ 0\ T$
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*:

$\Delta @ B :: \Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} U \implies \Delta[0 \mapsto_\tau U]_e @ \Gamma \vdash_{wf} T[||\Delta|| \mapsto_\tau U]_\tau$
 $\forall (l, T) \in set (rT::rcdT). \Delta @ B :: \Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} U \implies$
 $\forall (l, T) \in set rT. \Delta[0 \mapsto_\tau U]_e @ \Gamma \vdash_{wf} T[||\Delta|| \mapsto_\tau U]_\tau$
 $\Delta @ B :: \Gamma \vdash_{wf} snd (fT::fldT) \implies \Gamma \vdash_{wf} U \implies$
 $\Delta[0 \mapsto_\tau U]_e @ \Gamma \vdash_{wf} snd fT[||\Delta|| \mapsto_\tau U]_\tau$
apply (*induct* T **and** rT **and** fT *arbitrary: Δ and Δ and Δ*)

```

    rule: liftT.induct liftrT.induct liftfT.induct)
  apply (rename-tac nat Δ)
  apply simp-all
  apply (rule conjI)
  apply (rule impI)
  apply (drule-tac Γ=Γ and Δ=Δ[0 ↦τ 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 ||Δ|| ≤ nat - Suc 0)
  apply (rename-tac nata)
  apply (subgoal-tac nat - Suc ||Δ|| = 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 Δ)
  apply (erule well-formed-cases)
  apply (rule wf-all)
  apply simp
  apply (thin-tac ∧x. PROP P x for P :: - ⇒ prop)
  apply (drule-tac x=TVarB type1 :: Δ 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 ∀x ∈ S. P x for S P)
  apply (drule bpspec)
  apply assumption
  apply simp
  apply (simp add: split-paired-all)
  done

```

theorem *wf-dec*: $\Delta @ \Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} \downarrow_{\tau} \|\Delta\| \ 0 \ T$


```

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:  $\Delta @ B :: \Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} U \implies \Delta[0 \mapsto_{\tau} U]_e @ \Gamma \vdash_{wf}$ 
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  $\Rightarrow$  type  $\Rightarrow$  type  $\Rightarrow$  bool (-  $\vdash$  - <: - [50, 50, 50] 50)

```

where

```

  SA-Top:  $\Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} S \implies \Gamma \vdash S <: Top$ 
| SA-refl-TVar:  $\Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} TVar\ i \implies \Gamma \vdash TVar\ i <: TVar\ i$ 

```

| *SA-trans-TVar*: $\Gamma \langle i \rangle = [TVarB U] \implies$
 $\Gamma \vdash \uparrow_\tau (Suc\ i)\ 0\ U <: T \implies \Gamma \vdash TVar\ i <: T$
| *SA-arrow*: $\Gamma \vdash T_1 <: S_1 \implies \Gamma \vdash S_2 <: T_2 \implies \Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2$
| *SA-all*: $\Gamma \vdash T_1 <: S_1 \implies TVarB\ T_1 :: \Gamma \vdash S_2 <: T_2 \implies$
 $\Gamma \vdash (\forall <: S_1. S_2) <: (\forall <: T_1. T_2)$
| *SA-Rcd*: $\Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} RcdT\ fs \implies unique\ fs' \implies$
 $\forall (l, T) \in set\ fs'. \exists S. (l, S) \in set\ fs \wedge \Gamma \vdash S <: T \implies \Gamma \vdash RcdT\ fs <: RcdT\ fs'$

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 $H': \Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} U \implies P$
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
 $\Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} T \implies \Gamma \vdash T <: T$
 $\Gamma \vdash_{wf} \implies \forall (l::name, T) \in set\ fTs. \Gamma \vdash_{wf} T \longrightarrow \Gamma \vdash T <: T$
 $\Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} snd\ (fT::fldT) \implies \Gamma \vdash snd\ fT <: snd\ fT$
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 $wf: \Gamma \vdash_{wfB} B$
shows $\uparrow_e\ 1\ 0\ \Delta @ B :: \Gamma \vdash \uparrow_\tau\ 1\ \|\Delta\| P <: \uparrow_\tau\ 1\ \|\Delta\| Q$ **using** H
proof (*induct $\Delta @ \Gamma P Q$ arbitrary: Δ*)
case *SA-Top*
with *wf show ?case*
by (*auto intro: subtyping.SA-Top wfE-weaken wf-weaken*)
next
case *SA-refl-TVar*

```

with wf show ?case
  by (auto intro!; subtyping.SA-refl-TVar wfE-weaken dest: wf-weaken)
next
case (SA-trans-TVar i U T)
thus ?case
proof (cases i < ||Δ||)
  case True
  with SA-trans-TVar
  have ( $\uparrow_e 1 0 \Delta @ B :: \Gamma$ )(i) =  $[TVarB (\uparrow_\tau 1 (||\Delta|| - Suc\ i) U)]$ 
  by simp
  moreover from True SA-trans-TVar
  have  $\uparrow_e 1 0 \Delta @ B :: \Gamma \vdash$ 
     $\uparrow_\tau (Suc\ i) 0 (\uparrow_\tau 1 (||\Delta|| - Suc\ i) U) <: \uparrow_\tau 1 ||\Delta|| T$ 
  by simp
  ultimately have  $\uparrow_e 1 0 \Delta @ B :: \Gamma \vdash TVar\ i <: \uparrow_\tau 1 ||\Delta|| T$ 
  by (rule subtyping.SA-trans-TVar)
  with True show ?thesis by simp
next
  case False
  then have  $Suc\ i - ||\Delta|| = Suc\ (i - ||\Delta||)$  by arith
  with False SA-trans-TVar have ( $\uparrow_e 1 0 \Delta @ B :: \Gamma$ )(Suc i) =  $[TVarB U]$ 
  by simp
  moreover from False SA-trans-TVar
  have  $\uparrow_e 1 0 \Delta @ B :: \Gamma \vdash \uparrow_\tau (Suc\ (Suc\ i)) 0 U <: \uparrow_\tau 1 ||\Delta|| T$ 
  by simp
  ultimately have  $\uparrow_e 1 0 \Delta @ B :: \Gamma \vdash TVar\ (Suc\ i) <: \uparrow_\tau 1 ||\Delta|| T$ 
  by (rule subtyping.SA-trans-TVar)
  with False show ?thesis by simp
qed
next
  case SA-arrow
  thus ?case by simp (iprover intro: subtyping.SA-arrow)
next
  case (SA-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  $\uparrow_e (Suc\ 0) 0 \Delta @ B :: \Gamma \vdash_{wf}$  by simp (rule wfE-weaken)
  moreover from  $\langle \Delta @ \Gamma \vdash_{wf} RcdT\ fs \rangle$ 
  have  $\uparrow_e (Suc\ 0) 0 \Delta @ B :: \Gamma \vdash_{wf} \uparrow_\tau (Suc\ 0) ||\Delta|| (RcdT\ fs)$ 
  by (rule wf-weaken)
  hence  $\uparrow_e (Suc\ 0) 0 \Delta @ B :: \Gamma \vdash_{wf} RcdT (\uparrow_{r\tau} (Suc\ 0) ||\Delta|| fs)$  by simp
  moreover from SA-Rcd have unique ( $\uparrow_{r\tau} (Suc\ 0) ||\Delta|| fs'$ ) by simp
  moreover have  $\forall (l, T) \in set (\uparrow_{r\tau} (Suc\ 0) ||\Delta|| fs')$ .
     $\exists S. (l, S) \in set (\uparrow_{r\tau} (Suc\ 0) ||\Delta|| fs) \wedge \uparrow_e (Suc\ 0) 0 \Delta @ B :: \Gamma \vdash S <: T$ 
  proof (rule ballpI)
    fix l T
    assume  $(l, T) \in set (\uparrow_{r\tau} (Suc\ 0) ||\Delta|| fs')$ 

```

then obtain T' **where** $(l, T') \in \text{set } fs'$ **and** $T: T = \uparrow_\tau (Suc\ 0) \parallel \Delta \parallel T'$
by (*blast dest: liftrT-setD*)
with SA-Rcd obtain S **where**
 $lS: (l, S) \in \text{set } fs$
and $ST: \uparrow_e (Suc\ 0)\ 0\ \Delta\ @\ B :: \Gamma \vdash \uparrow_\tau (Suc\ 0) \parallel \Delta \parallel S <: \uparrow_\tau (Suc\ 0) \parallel \Delta \parallel$
 $(T[\parallel \Delta \parallel \mapsto_\tau Top]_\tau)$
by *fastforce*
with T **have** $\uparrow_e (Suc\ 0)\ 0\ \Delta\ @\ B :: \Gamma \vdash \uparrow_\tau (Suc\ 0) \parallel \Delta \parallel S <: \uparrow_\tau (Suc\ 0) \parallel \Delta \parallel$
 T'
by *simp*
moreover from lS **have** $(l, \uparrow_\tau (Suc\ 0) \parallel \Delta \parallel S) \in \text{set } (\uparrow_{r\tau} (Suc\ 0) \parallel \Delta \parallel fs)$
by (*rule liftrT-set*)
moreover note T
ultimately show $\exists S. (l, S) \in \text{set } (\uparrow_{r\tau} (Suc\ 0) \parallel \Delta \parallel fs) \wedge \uparrow_e (Suc\ 0)\ 0\ \Delta\ @\ B$
 $:: \Gamma \vdash S <: T$
by *auto*
qed
ultimately have $\uparrow_e (Suc\ 0)\ 0\ \Delta\ @\ B :: \Gamma \vdash RcdT (\uparrow_{r\tau} (Suc\ 0) \parallel \Delta \parallel fs) <:$
 $RcdT (\uparrow_{r\tau} (Suc\ 0) \parallel \Delta \parallel fs^\wedge)$
by (*rule subtyping.SA-Rcd*)
thus *?case* **by** *simp*
qed

lemma subtype-weaken': — A.2
 $\Gamma \vdash P <: Q \implies \Delta @ \Gamma \vdash_{wf} \implies \Delta @ \Gamma \vdash \uparrow_\tau \parallel \Delta \parallel 0\ P <: \uparrow_\tau \parallel \Delta \parallel 0\ Q$
apply (*induct* Δ)
apply *simp-all*
apply (*erule well-formedE-cases*)
apply *simp*
apply (*erule tac B=a and* $\Gamma=\Delta @ \Gamma$ **in** *subtype-weaken [of [], simplified]*)
apply *simp-all*
done

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

lemma subtype-trans: — A.3
 $\Gamma \vdash S <: Q \implies \Gamma \vdash Q <: T \implies \Gamma \vdash S <: T$
 $\Delta @ TVarB\ Q :: \Gamma \vdash M <: N \implies \Gamma \vdash P <: Q \implies$
 $\Delta @ TVarB\ P :: \Gamma \vdash M <: N$
using *wf-measure-size*
proof (*induct Q arbitrary: $\Gamma\ S\ T\ \Delta\ P\ M\ N$ rule: wf-induct-rule*)
case (*less Q*)
 $\{$
fix $\Gamma\ S\ T\ Q'$
assume $\Gamma \vdash S <: Q'$
then have $\Gamma \vdash Q' <: T \implies \text{size } Q = \text{size } Q' \implies \Gamma \vdash S <: T$
proof (*induct arbitrary: T*)

```

case SA-Top
from SA-Top(3) show ?case
  by cases (auto intro: subtyping.SA-Top SA-Top)
next
case SA-refl-TVar show ?case by fact
next
case SA-trans-TVar
thus ?case by (auto intro: subtyping.SA-trans-TVar)
next
case (SA-arrow  $\Gamma$   $T_1$   $S_1$   $S_2$   $T_2$ )
note SA-arrow' = SA-arrow
from SA-arrow(5) show ?case
proof cases
  case SA-Top
  with SA-arrow show ?thesis
  by (auto intro: subtyping.SA-Top wf-arrow elim: wf-subtypeE)
next
case (SA-arrow  $T_1'$   $T_2'$ )
from SA-arrow SA-arrow' have  $\Gamma \vdash S_1 \rightarrow S_2 <: T_1' \rightarrow T_2'$ 
  by (auto intro!: subtyping.SA-arrow intro: less(1) [of  $T_1$ ] less(1) [of  $T_2$ ])
with SA-arrow show ?thesis by simp
qed
next
case (SA-all  $\Gamma$   $T_1$   $S_1$   $S_2$   $T_2$ )
note SA-all' = SA-all
from SA-all(5) show ?case
proof cases
  case SA-Top
  with SA-all show ?thesis by (auto intro!:
    subtyping.SA-Top wf-all intro: wf-equallength elim: wf-subtypeE)
next
case (SA-all  $T_1'$   $T_2'$ )
from SA-all SA-all' have  $\Gamma \vdash T_1' <: S_1$ 
  by - (rule less(1), simp-all)
moreover from SA-all SA-all' have  $TVarB\ T_1' :: \Gamma \vdash S_2 <: T_2$ 
  by - (rule less(2) [of - []], simplified), simp-all)
with SA-all SA-all' have  $TVarB\ T_1' :: \Gamma \vdash S_2 <: T_2'$ 
  by - (rule less(1), simp-all)
ultimately have  $\Gamma \vdash (\forall <: S_1. S_2) <: (\forall <: T_1'. T_2')$ 
  by (rule subtyping.SA-all)
with SA-all show ?thesis by simp
qed
next
case (SA-Rcd  $\Gamma$   $fs_1$   $fs_2$ )
note SA-Rcd' = SA-Rcd
from SA-Rcd(5) show ?case
proof cases
  case SA-Top
  with SA-Rcd show ?thesis by (auto intro!: subtyping.SA-Top)

```

```

next
  case (SA-Rcd fs2')
  note  $\langle \Gamma \vdash_{wf} \rangle$ 
  moreover note  $\langle \Gamma \vdash_{wf} \text{RcdT } fs_1 \rangle$ 
  moreover note  $\langle \text{unique } fs_2' \rangle$ 
  moreover have  $\forall (l, T) \in \text{set } fs_2'. \exists S. (l, S) \in \text{set } fs_1 \wedge \Gamma \vdash S <: T$ 
  proof (rule ballpI)
    fix l T
    assume lT:  $(l, T) \in \text{set } fs_2'$ 
    with SA-Rcd obtain U where
      fs2:  $(l, U) \in \text{set } fs_2$  and UT:  $\Gamma \vdash U <: T$  by blast
    with SA-Rcd SA-Rcd' obtain S where
      fs1:  $(l, S) \in \text{set } fs_1$  and SU:  $\Gamma \vdash S <: U$  by blast
    from SA-Rcd SA-Rcd' fs2 have  $(U, Q) \in \text{measure size by simp}$ 
    hence  $\Gamma \vdash S <: T$  using SU UT by (rule less(1))
    with fs1 show  $\exists S. (l, S) \in \text{set } fs_1 \wedge \Gamma \vdash S <: T$  by blast
  qed
  ultimately have  $\Gamma \vdash \text{RcdT } fs_1 <: \text{RcdT } fs_2'$  by (rule subtyping.SA-Rcd)
  with SA-Rcd show ?thesis by simp
qed
qed
}
note tr = this
{
  case 1
  thus ?case using refl by (rule tr)
next
  case 2
  from 2(1) show  $\Delta @ \text{TVarB } P :: \Gamma \vdash M <: N$ 
  proof (induct  $\Delta @ \text{TVarB } Q :: \Gamma M N$  arbitrary:  $\Delta$ )
    case SA-Top
    with 2 show ?case by (auto intro!: subtyping.SA-Top
      intro: wf-equallength wfE-replace elim!: wf-subtypeE)
  next
    case SA-refl-TVar
    with 2 show ?case by (auto intro!: subtyping.SA-refl-TVar
      intro: wf-equallength wfE-replace elim!: wf-subtypeE)
  next
    case (SA-trans-TVar i U T)
    show ?case
    proof (cases  $i < \|\Delta\|$ )
      case True
      with SA-trans-TVar show ?thesis
      by (auto intro!: subtyping.SA-trans-TVar)
    next
      case False
      note False' = False
      show ?thesis
      proof (cases  $i = \|\Delta\|$ )

```

```

case True
from SA-trans-TVar have  $(\Delta @ [TVarB P]) @ \Gamma \vdash_{wf}$ 
  by (auto intro: wfE-replace elim!: wf-subtypeE)
with  $\langle \Gamma \vdash P <: Q \rangle$ 
  have  $(\Delta @ [TVarB P]) @ \Gamma \vdash \uparrow_{\tau} \|\Delta @ [TVarB P]\| 0 P <: \uparrow_{\tau} \|\Delta @$ 
 $[TVarB P]\| 0 Q$ 
  by (rule subtype-weaken')
with SA-trans-TVar True False have  $\Delta @ TVarB P :: \Gamma \vdash \uparrow_{\tau} (Suc \|\Delta\|)$ 
 $0 P <: T$ 
  by - (rule tr, simp+)
with True and False and SA-trans-TVar show ?thesis
  by (auto intro!: subtyping.SA-trans-TVar)
next
case False
with False' have  $i - \|\Delta\| = Suc (i - \|\Delta\| - 1)$  by arith
with False False' SA-trans-TVar show ?thesis
  by - (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  $\langle \Gamma \vdash P <: Q \rangle$  have  $\Gamma \vdash_{wf} P$  by (rule wf-subtypeE)
with SA-Rcd have  $\Delta @ TVarB P :: \Gamma \vdash_{wf}$ 
  by - (rule wfE-replace, simp+)
moreover from SA-Rcd have  $\Delta @ TVarB Q :: \Gamma \vdash_{wf} RcdT fs$  by simp
hence  $\Delta @ TVarB P :: \Gamma \vdash_{wf} RcdT fs$  by (rule wf-equallength) simp-all
moreover note ⟨unique fs'⟩
moreover from SA-Rcd
have  $\forall (l, T) \in set fs'. \exists S. (l, S) \in set fs \wedge \Delta @ TVarB P :: \Gamma \vdash S <: T$ 
  by blast
ultimately show ?case by (rule subtyping.SA-Rcd)
qed
qed

```

lemma *substT-subtype*: — A.10

assumes $H: \Delta @ TVarB Q :: \Gamma \vdash S <: T$

shows $\Gamma \vdash P <: Q \implies$

$\Delta[0 \mapsto_{\tau} P]_e @ \Gamma \vdash S[\|\Delta\| \mapsto_{\tau} P]_{\tau} <: T[\|\Delta\| \mapsto_{\tau} P]_{\tau}$

using H

apply (*induct* $\Delta @ TVarB Q :: \Gamma S T$ *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  $\|\Delta\| \leq i - \text{Suc } 0$ )
apply (rename-tac nat)
apply (subgoal-tac  $i - \text{Suc } \|\Delta\| = \text{nat}$ )
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  $\text{Suc } (\|\Delta\| - \text{Suc } 0) = \|\Delta\|$ )
apply (simp (no-asm-simp))
apply arith
apply (rule SA-arrow)
apply simp+
apply (rule SA-all)
apply simp
apply (drule-tac  $x = \text{TVarB } T_1 :: \Delta$  in meta-spec)
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  $H: \Delta @ \text{VarB } V :: \Gamma \vdash T <: U$ 
shows  $\downarrow_e 1 \ 0 \ \Delta @ \Gamma \vdash \downarrow_\tau 1 \ \|\Delta\| \ T <: \downarrow_\tau 1 \ \|\Delta\| \ U$ 
using H
apply (induct  $\Delta @ \text{VarB } V :: \Gamma \ T \ U$  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  $\|\Delta\| \leq i - \text{Suc } 0$ )
apply (rename-tac nat)
apply (subgoal-tac  $i - \text{Suc } \|\Delta\| = \text{nat}$ )
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 (drule-tac  $x = \text{TVarB } T_1 :: \Delta$  in meta-spec)

```

```

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 Δ , which can be thought of as a “flattened” list of types of the variables occurring in the pattern. Since typing environments are extended “to the left”, the bindings in Δ appear in reverse order.

inductive

```

ptyping :: pat  $\Rightarrow$  type  $\Rightarrow$  env  $\Rightarrow$  bool ( $\vdash - : - \Rightarrow -$  [50, 50, 50] 50)
and ptypings :: rpat  $\Rightarrow$  redT  $\Rightarrow$  env  $\Rightarrow$  bool ( $\vdash - [:] - \Rightarrow -$  [50, 50, 50] 50)

```

where

```

P-Var:  $\vdash PVar\ T : T \Rightarrow [VarB\ T]$ 
| P-Rcd:  $\vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow \vdash PRcd\ fps : RcdT\ fTs \Rightarrow \Delta$ 
| P-Nil:  $\vdash []\ [:]\ [] \Rightarrow []$ 
| P-Cons:  $\vdash p : T \Rightarrow \Delta_1 \Longrightarrow \vdash fps\ [:]\ fTs \Rightarrow \Delta_2 \Longrightarrow fps\langle l \rangle? = \perp \Longrightarrow$ 
 $\vdash ((l, p) :: fps)\ [:]\ ((l, T) :: fTs) \Rightarrow \uparrow_e\ \|\Delta_1\|\ 0\ \Delta_2\ @\ \Delta_1$ 

```

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

inductive

typing :: env ⇒ trm ⇒ type ⇒ bool (- ⊢ - : - [50, 50, 50] 50)

and *typings* :: env ⇒ rcd ⇒ rcdT ⇒ bool (- ⊢ - [:] - [50, 50, 50] 50)

where

T-Var: $\Gamma \vdash_{wf} U \implies \Gamma \langle i \rangle = \lfloor \text{VarB } U \rfloor \implies T = \uparrow_{\tau} (\text{Suc } i) \ 0 \ U \implies \Gamma \vdash \text{Var } i : T$
T-Abs: $\text{VarB } T_1 :: \Gamma \vdash t_2 : T_2 \implies \Gamma \vdash (\lambda:T_1. t_2) : T_1 \rightarrow \downarrow_{\tau} 1 \ 0 \ T_2$
T-App: $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \implies \Gamma \vdash t_2 : T_{11} \implies \Gamma \vdash t_1 \cdot t_2 : T_{12}$
T-TAbs: $\text{TVarB } T_1 :: \Gamma \vdash t_2 : T_2 \implies \Gamma \vdash (\lambda<:T_1. t_2) : (\forall<:T_1. T_2)$
T-TApp: $\Gamma \vdash t_1 : (\forall<:T_{11}. T_{12}) \implies \Gamma \vdash T_2 <: T_{11} \implies$
 $\Gamma \vdash t_1 \cdot_{\tau} T_2 : T_{12}[0 \mapsto_{\tau} T_2]_{\tau}$
T-Sub: $\Gamma \vdash t : S \implies \Gamma \vdash S <: T \implies \Gamma \vdash t : T$
T-Let: $\Gamma \vdash t_1 : T_1 \implies \vdash p : T_1 \Rightarrow \Delta \implies \Delta @ \Gamma \vdash t_2 : T_2 \implies$
 $\Gamma \vdash (\text{LET } p = t_1 \ \text{IN } t_2) : \downarrow_{\tau} \|\Delta\| \ 0 \ T_2$
T-Rcd: $\Gamma \vdash fs \ [:] \ fTs \implies \Gamma \vdash \text{Rcd } fs : \text{RcdT } fTs$
T-Proj: $\Gamma \vdash t : \text{RcdT } fTs \implies fTs \langle l \rangle? = \lfloor T \rfloor \implies \Gamma \vdash t..l : T$
T-Nil: $\Gamma \vdash_{wf} \implies \Gamma \vdash \lfloor \cdot \rfloor \ \lfloor \cdot \rfloor$
T-Cons: $\Gamma \vdash t : T \implies \Gamma \vdash fs \ [:] \ fTs \implies fs \langle l \rangle? = \perp \implies$
 $\Gamma \vdash (l, t) :: fs \ [:] \ (l, T) :: fTs$

theorem *wf-typeE1*:

$\Gamma \vdash t : T \implies \Gamma \vdash_{wf} t$

$\Gamma \vdash fs \ [:] \ fTs \implies \Gamma \vdash_{wf} fs$

by (*induct set: typing typings*) (*blast elim: well-formedE-cases*)+

theorem *wf-typeE2*:

$\Gamma \vdash t : T \implies \Gamma \vdash_{wf} T$

$\Gamma' \vdash fs \ [:] \ fTs \implies (\forall (l, T) \in \text{set } fTs. \Gamma' \vdash_{wf} T) \wedge$

$\text{unique } fTs \wedge (\forall l. (fs \langle l \rangle? = \perp) = (fTs \langle l \rangle? = \perp))$

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 - - - λx y z. True, simplified True-simps, consumes 1,*
case-names P-Var P-Rcd]

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

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

lemma *narrow-type*: — A.7

$\Delta @ TVarB Q :: \Gamma \vdash t : T \implies$
 $\Gamma \vdash P <: Q \implies \Delta @ TVarB P :: \Gamma \vdash t : T$
 $\Delta @ TVarB Q :: \Gamma \vdash ts [:] Ts \implies$
 $\Gamma \vdash P <: Q \implies \Delta @ TVarB P :: \Gamma \vdash ts [:] Ts$
apply (*induct* $\Delta @ TVarB Q :: \Gamma \vdash t : T$ **and** $\Delta @ TVarB Q :: \Gamma \vdash ts Ts$
arbitrary: Δ and Δ set: typing typlings)
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 (drule-tac x=VarB T1 :: Δ in meta-spec)
apply simp
apply (rule-tac T11=T11 in T-App)
apply simp+
apply (rule T-TAbs)
apply (drule-tac x=TVarB T1 :: Δ in meta-spec)
apply simp
apply (rule-tac T11=T11 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 (drule-tac x=Δ @ Δ' in meta-spec, drule meta-mp, rule refl)
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 $(l, T) \in set fTs \implies \exists t. fs(l)? = [t] \wedge \Gamma \vdash t : T$
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** $\Gamma \vdash_{wf} t$ **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 $\Gamma \vdash T <: U \rightarrow U' \Longrightarrow$
 $(\bigwedge S'. \Gamma \vdash U <: S \Longrightarrow \text{VarB } S :: \Gamma \vdash s : S' \Longrightarrow$
 $\Gamma \vdash \downarrow_{\tau} 1 0 S' <: U' \Longrightarrow P) \Longrightarrow P$
using H
proof (*induct* $\Gamma \lambda:S. s T$ *arbitrary*: $U U' S s P$)
case ($T\text{-Abs } T_1 \Gamma t_2 T_2$)
from $\langle \Gamma \vdash T_1 \rightarrow \downarrow_{\tau} 1 0 T_2 <: U \rightarrow U' \rangle$
obtain $ty1: \Gamma \vdash U <: T_1$ **and** $ty2: \Gamma \vdash \downarrow_{\tau} 1 0 T_2 <: U'$
by *cases simp-all*
from $ty1 \langle \text{VarB } T_1 :: \Gamma \vdash t_2 : T_2 \rangle ty2$
show $?case$ **by** (*rule T-Abs*)
next
case ($T\text{-Sub } \Gamma S' T$)
from $\langle \Gamma \vdash S' <: T \rangle$ **and** $\langle \Gamma \vdash T <: U \rightarrow U' \rangle$
have $\Gamma \vdash S' <: U \rightarrow U'$ **by** (*rule subtype-trans(1)*)
then show $?case$
by (*rule T-Sub*) (*rule T-Sub(5)*)
qed

lemma *Abs-type'*:
assumes $H: \Gamma \vdash (\lambda:S. s) : U \rightarrow U'$
and $R: \bigwedge S'. \Gamma \vdash U <: S \Longrightarrow \text{VarB } S :: \Gamma \vdash s : S' \Longrightarrow$
 $\Gamma \vdash \downarrow_{\tau} 1 0 S' <: U' \Longrightarrow P$
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 $\Gamma \vdash T <: (\forall<:U. U') \Longrightarrow$
 $(\bigwedge S'. \Gamma \vdash U <: S \Longrightarrow \text{TVarB } U :: \Gamma \vdash s : S' \Longrightarrow$
 $\text{TVarB } U :: \Gamma \vdash S' <: U' \Longrightarrow P) \Longrightarrow P$
using H
proof (*induct* $\Gamma \lambda<:S. s T$ *arbitrary*: $U U' S s P$)
case ($T\text{-TAbs } T_1 \Gamma t_2 T_2$)
from $\langle \Gamma \vdash (\forall<:T_1. T_2) <: (\forall<:U. U') \rangle$
obtain $ty1: \Gamma \vdash U <: T_1$ **and** $ty2: \text{TVarB } U :: \Gamma \vdash T_2 <: U'$
by *cases simp-all*
from $\langle \text{TVarB } T_1 :: \Gamma \vdash t_2 : T_2 \rangle$
have $\text{TVarB } U :: \Gamma \vdash t_2 : T_2$ **using** $ty1$
by (*rule narrow-type* [*of []*, *simplified*])
with $ty1$ **show** $?case$ **using** $ty2$ **by** (*rule T-TAbs*)
next
case ($T\text{-Sub } \Gamma S' T$)
from $\langle \Gamma \vdash S' <: T \rangle$ **and** $\langle \Gamma \vdash T <: (\forall<:U. U') \rangle$
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 $R: \bigwedge S'. \Gamma \vdash U <: S \implies TVarB U :: \Gamma \vdash s : S' \implies$
 $TVarB U :: \Gamma \vdash S' <: U' \implies P$
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 $t = Rcd fs \implies \Gamma \vdash T <: RcdT fTs \implies$
 $\forall (l, U) \in set fTs. \exists u. fs\langle l \rangle? = [u] \wedge \Gamma \vdash u : U$
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 $\forall (l, U) \in set fTs. \exists u. fs\langle l \rangle? = [u] \wedge \Gamma \vdash u : U$
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,

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

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*:

$\Gamma \vdash \text{Rcd } fs : T \implies \Gamma \vdash T <: \text{RcdT } fTs \implies$

$\Gamma \vdash \text{map } (\lambda(l, T). (l, \text{the } (fs\langle l \rangle?))) fTs [:] fTs$

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 $\Gamma \vdash \text{RcdT } ((a, b) :: list) <: \text{RcdT } list$*)

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 $H: \Gamma \vdash \text{Rcd } fs : \text{RcdT } fTs$

shows $\Gamma \vdash \text{map } (\lambda(l, T). (l, \text{the } (fs\langle l \rangle?))) fTs [:] fTs$

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 ptyplings*) *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 ptyplings*)

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: Δ and Δ 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 (*drule-tac x=VarB T₁ :: Δ in meta-spec*)

```

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 (drule-tac  $x=TVarB$   $T_1 :: \Delta$  in meta-spec)
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  $x=\Delta$  @  $\Delta'$  in meta-spec)
apply (drule meta-mp)
apply simp
apply simp
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)
 $\Gamma \vdash t : T \implies \Delta @ \Gamma \vdash_{wf} \implies \Delta @ \Gamma \vdash \uparrow \|\Delta\|$  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(1) [of [], simplified])
apply simp+
done

```

The substitution lemmas are now proved by mutual induction on the deriva-

tions of the typing derivations for terms and lists of fields.

lemma *subst-ptying*:

$\vdash p : T \Rightarrow \Delta \Longrightarrow \vdash p[k \mapsto_{\tau} U]_p : T[k \mapsto_{\tau} U]_{\tau} \Rightarrow \Delta[k \mapsto_{\tau} U]_e$
 $\vdash fps [:] fTs \Rightarrow \Delta \Longrightarrow \vdash fps[k \mapsto_{\tau} U]_{rp} [:] fTs[k \mapsto_{\tau} U]_{r\tau} \Rightarrow \Delta[k \mapsto_{\tau} U]_e$
apply (*induct set: ptying ptyings*)
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

$\Delta @ VarB U :: \Gamma \vdash t : T \Longrightarrow \Gamma \vdash u : U \Longrightarrow$
 $\downarrow_e 1 0 \Delta @ \Gamma \vdash t[\|\Delta\| \mapsto u] : \downarrow_{\tau} 1 \|\Delta\| T$
 $\Delta @ VarB U :: \Gamma \vdash fs [:] fTs \Longrightarrow \Gamma \vdash u : U \Longrightarrow$
 $\downarrow_e 1 0 \Delta @ \Gamma \vdash fs[\|\Delta\| \mapsto u]_r [:] \downarrow_{r\tau} 1 \|\Delta\| fTs$
apply (*induct $\Delta @ VarB U :: \Gamma \vdash t T$ and $\Delta @ VarB U :: \Gamma \vdash fs fTs$*
arbitrary: Δ and Δ 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 (drule-tac  $x = \text{VarB } T_1 :: \Delta$  in meta-spec)
apply (rule T-Abs [THEN T-eq])
apply simp
apply (simp add: substT-substT [symmetric])
apply simp
apply (rule-tac  $T_{11} = T_{11} [ \|\Delta\| \mapsto_\tau \text{Top} ]_\tau$  in T-App)
apply simp+
apply (rule T-TAbs)
apply (drule-tac  $x = \text{TVarB } T_1 :: \Delta$  in meta-spec)
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  $S = S [ \|\Delta\| \mapsto_\tau \text{Top} ]_\tau$  in T-Sub)
apply simp
apply simp
apply (rule subst-subtype [simplified])
apply assumption
— records
apply simp
apply (drule meta-spec)+
apply (drule meta-mp, rule refl)+
apply (drule-tac  $\Gamma = \Delta' [ 0 \mapsto_\tau \text{Top} ]_e @ \Gamma$  in T-Let)
apply (erule subst-ptying)
apply simp
apply (simp add: ac-simps)
apply simp
apply (rule T-Rcd)
apply simp
apply simp
apply (rule-tac  $fTs = fTs [ \|\Delta\| \mapsto_\tau \text{Top} ]_{r\tau}$  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

$$\Delta @ \text{TVarB } Q :: \Gamma \vdash t : T \implies \Gamma \vdash P <: Q \implies \\ \Delta [0 \mapsto_\tau P]_e @ \Gamma \vdash t [\|\Delta\| \mapsto_\tau P] : T [\|\Delta\| \mapsto_\tau P]_\tau$$

$\Delta @ TVarB Q :: \Gamma \vdash fs [:] fTs \implies \Gamma \vdash P <: Q \implies$
 $\Delta[0 \mapsto_{\tau} P]_e @ \Gamma \vdash fs[||\Delta|| \mapsto_{\tau} P]_r [:] fTs[||\Delta|| \mapsto_{\tau} P]_{r\tau}$
apply (*induct* $\Delta @ TVarB Q :: \Gamma t T$ **and** $\Delta @ TVarB Q :: \Gamma fs fTs$
arbitrary: Δ and Δ 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 (*drule-tac* $x = TVarB\ T_1 :: \Delta$ **in** *meta-spec*)
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 (*drule-tac* $x = TVarB\ T_1 :: \Delta$ **in** *meta-spec*)
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 meta-spec)+
apply (drule meta-mp, rule refl)+
apply (drule-tac  $\Gamma = \Delta' [0 \mapsto_{\tau} P]_e @ \Gamma$  in T-Let)
apply (erule subst-ptying)
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: $(\lambda:T. t) \in \text{value}$

| *TAbs*: $(\lambda<:T. t) \in \text{value}$

| *Rcd*: $\forall (l, t) \in \text{set } fs. t \in \text{value} \implies Rcd\ fs \in \text{value}$

In order to formalize the evaluation rule for *LET*, we introduce another relation $\vdash p \triangleright t \Rightarrow ts$ expressing that a pattern p matches a term t . The relation also yields a list of terms ts corresponding to the variables in the pattern. The relation is defined simultaneously with another relation $\vdash fps [\triangleright] fs \Rightarrow ts$ for matching a list of field patterns fps against a list of fields fs :

inductive

match :: *pat* \Rightarrow *trm* \Rightarrow *trm list* \Rightarrow *bool* ($\vdash - \triangleright - \Rightarrow - [50, 50, 50] 50$)

and *matches* :: *rpat* \Rightarrow *rcd* \Rightarrow *trm list* \Rightarrow *bool* ($\vdash - [\triangleright] - \Rightarrow - [50, 50, 50] 50$)

where

M-PVar: $\vdash PVar\ T \triangleright t \Rightarrow [t]$

| *M-Rcd*: $\vdash fps [\triangleright] fs \Rightarrow ts \implies \vdash PRcd\ fps \triangleright Rcd\ fs \Rightarrow ts$

| *M-Nil*: $\vdash [] [\triangleright] fs \Rightarrow []$

| *M-Cons*: $fs \langle l \rangle ? = [t] \implies \vdash p \triangleright t \Rightarrow ts \implies \vdash fps [\triangleright] fs \Rightarrow us \implies$

$\vdash (l, p) :: fps [\triangleright] fs \Rightarrow ts @ us$

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

inductive

eval :: *trm* ⇒ *trm* ⇒ *bool* (**infixl** \mapsto 50)
and *evals* :: *rcd* ⇒ *rcd* ⇒ *bool* (**infixl** \mapsto 50)

where

E-Abs: $v_2 \in \text{value} \implies (\lambda:T_{11}. t_{12}) \cdot v_2 \mapsto t_{12}[0 \mapsto v_2]$
| *E-TAbs*: $(\lambda<:T_{11}. t_{12}) \cdot_{\tau} T_2 \mapsto t_{12}[0 \mapsto_{\tau} T_2]$
| *E-App1*: $t \mapsto t' \implies t \cdot u \mapsto t' \cdot u$
| *E-App2*: $v \in \text{value} \implies t \mapsto t' \implies v \cdot t \mapsto v \cdot t'$
| *E-TApp*: $t \mapsto t' \implies t \cdot_{\tau} T \mapsto t' \cdot_{\tau} T$
| *E-LetV*: $v \in \text{value} \implies \vdash p \triangleright v \Rightarrow ts \implies (\text{LET } p = v \text{ IN } t) \mapsto t[0 \mapsto_s ts]$
| *E-ProjRcd*: $fs\langle l \rangle? = [v] \implies v \in \text{value} \implies \text{Rcd } fs..l \mapsto v$
| *E-Proj*: $t \mapsto t' \implies t..l \mapsto t'..l$
| *E-Rcd*: $fs \mapsto fs' \implies \text{Rcd } fs \mapsto \text{Rcd } fs'$
| *E-Let*: $t \mapsto t' \implies (\text{LET } p = t \text{ IN } u) \mapsto (\text{LET } p = t' \text{ IN } u)$
| *E-hd*: $t \mapsto t' \implies (l, t) :: fs \mapsto (l, t') :: fs$
| *E-tl*: $v \in \text{value} \implies fs \mapsto fs' \implies (l, v) :: fs \mapsto (l, v) :: fs'$

The relation $t \mapsto t'$ is defined simultaneously with a relation $fs \mapsto fs'$ 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 - - $\lambda x y z. \text{True}$, *simplified True-simps*, *consumes 1*,
case-names M-Nil M-Cons]

lemmas *evals-induct* = *eval-evals.inducts*(2)

[of - - $\lambda x y. \text{True}$, *simplified True-simps*, *consumes 1*,
case-names E-hd E-tl]

lemma *matchs-mono*:

assumes $H: \vdash fps \triangleright fs \Rightarrow ts$
shows $fps\langle l \rangle? = \perp \implies \vdash fps \triangleright (l, t) :: fs \Rightarrow ts$
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 $\forall (l, p) \in \text{set } fps. fs\langle l \rangle? = fs'\langle l \rangle? \implies \vdash fps \triangleright fs' \Rightarrow ts$
using *H*
apply (*induct rule: matchs-induct*)

apply (rule *M-Nil*)
apply (rule *M-Cons*)
apply *auto*
done

lemma *reorder-eq*:

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

lemma *matchs-reorder*:

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

lemma *matchs-reorder'*:

$\vdash \text{fps } [:] \text{fTs} \Rightarrow \Delta \Longrightarrow \forall (l, U) \in \text{set fTs}. \exists u. \text{fs}\langle l \rangle? = \lfloor u \rfloor \Longrightarrow$
 $\vdash \text{fps } [\triangleright] \text{map } (\lambda(l, T). (l, \text{the } (\text{fs}\langle l \rangle?))) \text{fTs} \Rightarrow \text{ts} \Longrightarrow \vdash \text{fps } [\triangleright] \text{fs} \Rightarrow \text{ts}$
by (rule *matchs-eq* [*OF - reorder-eq* [*THEN ball-eq-sym*]], *assumption+*)

theorem *matchs-tl*:

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

theorem *match-length*:

$\vdash p \triangleright t \Rightarrow \text{ts} \Longrightarrow \vdash p : T \Rightarrow \Delta \Longrightarrow \|\text{ts}\| = \|\Delta\|$
 $\vdash \text{fps } [\triangleright] \text{ft} \Rightarrow \text{ts} \Longrightarrow \vdash \text{fps } [:] \text{fTs} \Rightarrow \Delta \Longrightarrow \|\text{ts}\| = \|\Delta\|$
by (induct *arbitrary*: $T\ \Delta$ **and** *fTs* Δ *set*: *match matchs*)
(*erule ptypling.cases ptyplings.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

$\vdash p : T_1 \Rightarrow \Delta \Longrightarrow \Gamma_2 \vdash t_1 : T_1 \Longrightarrow$
 $\Gamma_1 @ \Delta @ \Gamma_2 \vdash t_2 : T_2 \Longrightarrow \vdash p \triangleright t_1 \Rightarrow \text{ts} \Longrightarrow$
 $\downarrow_e \|\Delta\| \ 0\ \Gamma_1 @ \Gamma_2 \vdash t_2 [\|\Gamma_1\| \mapsto_s \text{ts}] : \downarrow_\tau \|\Delta\| \ \|\Gamma_1\| \ T_2$
 $\vdash \text{fps } [:] \text{fTs} \Rightarrow \Delta \Longrightarrow \Gamma_2 \vdash \text{fs } [:] \text{fTs} \Longrightarrow$

$\Gamma_1 @ \Delta @ \Gamma_2 \vdash t_2 : T_2 \implies \vdash \text{fps} [\triangleright] fs \Rightarrow ts \implies$
 $\downarrow_e \|\Delta\| 0 \Gamma_1 @ \Gamma_2 \vdash t_2[\|\Gamma_1\| \mapsto_s ts] : \downarrow_\tau \|\Delta\| \|\Gamma_1\| T_2$
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: ptying ptyings*)
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* (β) **have** $ts = [t_1]$ **by cases simp-all**
ultimately show *?case* **by simp**
next
case (*P-Rcd $\text{fps } fTs \Delta \Gamma_1 \Gamma_2 t_1 t_2 T_2 ts$*)
from *P-Rcd*(5) **obtain** *fs* **where**
 $t_1 : t_1 = \text{Rcd } fs$ **and** $\text{fps} : \vdash \text{fps} [\triangleright] fs \Rightarrow ts$ **by cases simp-all**
with *P-Rcd* **have** $fs : \Gamma_2 \vdash \text{Rcd } fs : \text{Rcd } T fTs$ **by simp**
hence $\Gamma_2 \vdash \text{map } (\lambda(l, T). (l, \text{the } (fs\langle l \rangle?))) fTs [\cdot] fTs$
by (*rule Rcd-type2'*)
moreover note *P-Rcd*(4)
moreover from *fs* **have** $\forall (l, U) \in \text{set } fTs. \exists u. fs\langle l \rangle? = [u] \wedge \Gamma_2 \vdash u : U$
by (*rule Rcd-type1'*)
hence $\forall (l, U) \in \text{set } fTs. \exists u. fs\langle l \rangle? = [u]$ **by blast**
with *P-Rcd*(1) **have** $\vdash \text{fps} [\triangleright] \text{map } (\lambda(l, T). (l, \text{the } (fs\langle l \rangle?))) fTs \Rightarrow ts$
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 $p T \Delta_1 \text{fps } fTs \Delta_2 l \Gamma_1 \Gamma_2 fs t_2 T_2 ts$*)
from *P-Cons*(8) **obtain** $t ts_1 ts_2$ **where**
 $t : fs\langle l \rangle? = [t]$ **and** $p : \vdash p \triangleright t \Rightarrow ts_1$ **and** $\text{fps} : \vdash \text{fps} [\triangleright] fs \Rightarrow ts_2$
and $ts : ts = ts_1 @ ts_2$ **by cases simp-all**
from *P-Cons*(6) t *fps* **obtain** fs' **where**
 $\text{fps}' : \vdash \text{fps} [\triangleright] (l, t) :: fs' \Rightarrow ts_2$ **and** $tT : \Gamma_2 \vdash t : T$ **and** $fs' : \Gamma_2 \vdash fs' [\cdot] fTs$
and $l : fs'\langle l \rangle? = \perp$ **by cases auto**
from *P-Cons* **have** $(\Gamma_1 @ \uparrow_e \|\Delta_1\| 0 \Delta_2) @ \Delta_1 @ \Gamma_2 \vdash t_2 : T_2$ **by simp**
with tT **have** $ts_1 : \downarrow_e \|\Delta_1\| 0 (\Gamma_1 @ \uparrow_e \|\Delta_1\| 0 \Delta_2) @ \Gamma_2 \vdash$
 $t_2[\|\Gamma_1\| @ \uparrow_e \|\Delta_1\| 0 \Delta_2] \mapsto_s ts_1] : \downarrow_\tau \|\Delta_1\| \|\Gamma_1\| @ \uparrow_e \|\Delta_1\| 0 \Delta_2\| T_2$
using p **by** (*rule P-Cons*)
from fps' *P-Cons*(5) **have** $\vdash \text{fps} [\triangleright] fs' \Rightarrow ts_2$ **by** (*rule matchs-tl*)
with $fs' ts_1$ [*simplified*]
have $\downarrow_e \|\Delta_2\| 0 (\downarrow_e \|\Delta_1\| \|\Delta_2\| \Gamma_1) @ \Gamma_2 \vdash t_2[\|\Gamma_1\| + \|\Delta_2\| \mapsto_s ts_1][\downarrow_e \|\Delta_1\|$
 $\|\Delta_2\| \Gamma_1\| \mapsto_s ts_2] :$
 $\downarrow_\tau \|\Delta_2\| \|\downarrow_e \|\Delta_1\| \|\Delta_2\| \Gamma_1\| (\downarrow_\tau \|\Delta_1\| (\|\Gamma_1\| + \|\Delta_2\|)) T_2)$
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 $H: fs \mapsto fs'$
shows $(fs \langle l \rangle? = \perp) = (fs' \langle l \rangle? = \perp)$ **using** H
by (*induct rule: evals-induct*) *simp-all*

theorem *preservation*: — A.20
 $\Gamma \vdash t : T \implies t \mapsto t' \implies \Gamma \vdash t' : T$
 $\Gamma \vdash fs \text{ [:] } fTs \implies fs \mapsto fs' \implies \Gamma \vdash fs' \text{ [:] } fTs$
proof (*induct arbitrary: t' and fs' set: typing typings*)
case ($T\text{-Var } \Gamma \ i \ U \ T \ t'$)
from $\langle \text{Var } i \mapsto t' \rangle$
show *?case by cases*

next
case ($T\text{-Abs } T_1 \ \Gamma \ t_2 \ T_2 \ t'$)
from $\langle (\lambda: T_1. t_2) \mapsto t' \rangle$
show *?case by cases*

next
case ($T\text{-App } \Gamma \ t_1 \ T_{11} \ T_{12} \ t_2 \ t'$)
from $\langle t_1 \cdot t_2 \mapsto t' \rangle$
show *?case*
proof *cases*
case ($E\text{-Abs } T_{11}' \ t_{12}$)
with $T\text{-App}$ **have** $\Gamma \vdash (\lambda: T_{11}'. t_{12}) : T_{11} \rightarrow T_{12}$ **by** *simp*
then obtain S'
where $T_{11}: \Gamma \vdash T_{11} <: T_{11}'$
and $t_{12}: \text{VarB } T_{11}' :: \Gamma \vdash t_{12} : S'$
and $S': \Gamma \vdash S'[0 \mapsto_{\tau} \text{Top}]_{\tau} <: T_{12}$ **by** (*rule Abs-type' [simplified]*) *blast*
from $\langle \Gamma \vdash t_2 : T_{11} \rangle$
have $\Gamma \vdash t_2 : T_{11}'$ **using** T_{11} **by** (*rule T-Sub*)
with t_{12} **have** $\Gamma \vdash t_{12}[0 \mapsto t_2] : S'[0 \mapsto_{\tau} \text{Top}]_{\tau}$
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 $E\text{-Abs}$ **show** *?thesis by simp*

next
case ($E\text{-App1 } t''$)
from $\langle t_1 \mapsto t'' \rangle$
have $\Gamma \vdash t'' : T_{11} \rightarrow T_{12}$ **by** (*rule T-App*)
hence $\Gamma \vdash t'' \cdot t_2 : T_{12}$ **using** $\langle \Gamma \vdash t_2 : T_{11} \rangle$
by (*rule typing-typings.T-App*)
with $E\text{-App1}$ **show** *?thesis by simp*

next
case ($E\text{-App2 } t''$)
from $\langle t_2 \mapsto t'' \rangle$
have $\Gamma \vdash t'' : T_{11}$ **by** (*rule T-App*)
with $T\text{-App}(1)$ **have** $\Gamma \vdash t_1 \cdot t'' : T_{12}$
by (*rule typing-typings.T-App*)
with $E\text{-App2}$ **show** *?thesis by simp*

qed
next

```

case ( $T\text{-TAbs } T_1 \Gamma t_2 T_2 t'$ )
from  $\langle \lambda <: T_1. t_2 \mapsto t' \rangle$ 
show  $?case$  by cases
next
case ( $T\text{-TApp } \Gamma t_1 T_{11} T_{12} T_2 t'$ )
from  $\langle t_1 \cdot_\tau T_2 \mapsto t' \rangle$ 
show  $?case$ 
proof cases
  case ( $E\text{-TAbs } T_{11}' t_{12}$ )
  with  $T\text{-TApp}$  have  $\Gamma \vdash (\lambda <: T_{11}'. t_{12}) : (\forall <: T_{11}. T_{12})$  by simp
  then obtain  $S'$ 
    where  $TVarB T_{11} :: \Gamma \vdash t_{12} : S'$ 
    and  $TVarB T_{11} :: \Gamma \vdash S' <: T_{12}$  by (rule TAbs-type') blast
  hence  $TVarB T_{11} :: \Gamma \vdash t_{12} : T_{12}$  by (rule T-Sub)
  hence  $\Gamma \vdash t_{12}[0 \mapsto_\tau T_2] : T_{12}[0 \mapsto_\tau T_2]_\tau$  using  $T\text{-TApp}(3)$ 
    by (rule substT-type [where  $\Delta = []$ , simplified])
  with  $E\text{-TAbs}$  show  $?thesis$  by simp
next
  case ( $E\text{-TApp } t''$ )
  from  $\langle t_1 \mapsto t'' \rangle$ 
  have  $\Gamma \vdash t'' : (\forall <: T_{11}. T_{12})$  by (rule T-TApp)
  hence  $\Gamma \vdash t'' \cdot_\tau T_2 : T_{12}[0 \mapsto_\tau T_2]_\tau$  using  $\langle \Gamma \vdash T_2 <: T_{11} \rangle$ 
    by (rule typing-typings.T-TApp)
  with  $E\text{-TApp}$  show  $?thesis$  by simp
qed
next
case ( $T\text{-Sub } \Gamma t S T t'$ )
from  $\langle t \mapsto t' \rangle$ 
have  $\Gamma \vdash t' : S$  by (rule T-Sub)
then show  $?case$  using  $\langle \Gamma \vdash S <: T \rangle$ 
    by (rule typing-typings.T-Sub)
next
case ( $T\text{-Let } \Gamma t_1 T_1 p \Delta t_2 T_2 t'$ )
from  $\langle (LET\ p = t_1\ IN\ t_2) \mapsto t' \rangle$ 
show  $?case$ 
proof cases
  case ( $E\text{-LetV } ts$ )
  from  $T\text{-Let } (3,1,4) \langle \vdash p \triangleright t_1 \Rightarrow ts \rangle$ 
  have  $\Gamma \vdash t_2[0 \mapsto_s ts] : \downarrow_\tau \|\Delta\| 0 T_2$ 
    by (rule match-type(1) [of - - - - [], simplified])
  with  $E\text{-LetV}$  show  $?thesis$  by simp
next
  case ( $E\text{-Let } t''$ )
  from  $\langle t_1 \mapsto t'' \rangle$ 
  have  $\Gamma \vdash t'' : T_1$  by (rule T-Let)
  hence  $\Gamma \vdash (LET\ p = t''\ IN\ t_2) : \downarrow_\tau \|\Delta\| 0 T_2$  using  $T\text{-Let}(3,4)$ 
    by (rule typing-typings.T-Let)
  with  $E\text{-Let}$  show  $?thesis$  by simp
qed

```

next
case ($T\text{-Rcd } \Gamma \text{ } fs \text{ } fTs \text{ } t'$)
from $\langle Rcd \text{ } fs \text{ } \mapsto \text{ } t' \rangle$
obtain fs' **where** $t' : t' = Rcd \text{ } fs'$ **and** $fs : fs \text{ } [\mapsto] \text{ } fs'$
by *cases simp-all*
from fs **have** $\Gamma \vdash fs' \text{ } [:] \text{ } fTs$ **by** (*rule T-Rcd*)
hence $\Gamma \vdash Rcd \text{ } fs' : RcdT \text{ } fTs$ **by** (*rule typing-typings.T-Rcd*)
with t' **show** $?case$ **by** *simp*

next
case ($T\text{-Proj } \Gamma \text{ } t \text{ } fTs \text{ } l \text{ } T \text{ } t'$)
from $\langle t..l \text{ } \mapsto \text{ } t' \rangle$
show $?case$
proof *cases*
case ($E\text{-ProjRcd } fs$)
with $T\text{-Proj}$ **have** $\Gamma \vdash Rcd \text{ } fs : RcdT \text{ } fTs$ **by** *simp*
hence $\forall (l, U) \in set \text{ } fTs. \exists u. fs \langle l \rangle ? = [u] \wedge \Gamma \vdash u : U$
by (*rule Rcd-type1*)
with $E\text{-ProjRcd } T\text{-Proj}$ **show** $?thesis$ **by** (*fastforce dest: assoc-set*)

next
case ($E\text{-Proj } t''$)
from $\langle t \text{ } \mapsto \text{ } t'' \rangle$
have $\Gamma \vdash t'' : RcdT \text{ } fTs$ **by** (*rule T-Proj*)
hence $\Gamma \vdash t''..l : T$ **using** $T\text{-Proj}(3)$
by (*rule typing-typings.T-Proj*)
with $E\text{-Proj}$ **show** $?thesis$ **by** *simp*

qed

next
case ($T\text{-Nil } \Gamma \text{ } fs'$)
from $\langle [] \text{ } [\mapsto] \text{ } fs' \rangle$
show $?case$ **by** *cases*

next
case ($T\text{-Cons } \Gamma \text{ } t \text{ } T \text{ } fs \text{ } fTs \text{ } l \text{ } fs'$)
from $\langle (l, t) :: fs \text{ } [\mapsto] \text{ } fs' \rangle$
show $?case$
proof *cases*
case ($E\text{-hd } t'$)
from $\langle t \text{ } \mapsto \text{ } t' \rangle$
have $\Gamma \vdash t' : T$ **by** (*rule T-Cons*)
hence $\Gamma \vdash (l, t') :: fs \text{ } [:] \text{ } (l, T) :: fTs$ **using** $T\text{-Cons}(3,5)$
by (*rule typing-typings.T-Cons*)
with $E\text{-hd}$ **show** $?thesis$ **by** *simp*

next
case ($E\text{-tl } fs''$)
note $fs = \langle fs \text{ } [\mapsto] \text{ } fs'' \rangle$
note $T\text{-Cons}(1)$
moreover from fs **have** $\Gamma \vdash fs'' \text{ } [:] \text{ } fTs$ **by** (*rule T-Cons*)
moreover from $fs \text{ } T\text{-Cons}$ **have** $fs'' \langle l \rangle ? = \perp$ **by** *simp*
ultimately have $\Gamma \vdash (l, t) :: fs'' \text{ } [:] \text{ } (l, T) :: fTs$
by (*rule typing-typings.T-Cons*)

```

    with E-tl show ?thesis by simp
  qed
qed

lemma Fun-canonical: — A.14(1)
  assumes ty:  $\square \vdash v : T_1 \rightarrow T_2$ 
  shows  $v \in \text{value} \implies \exists t S. v = (\lambda:S. t)$  using ty
proof (induct  $\square::\text{env } v : T_1 \rightarrow T_2$  arbitrary:  $T_1 T_2$  rule: typing-induct)
  case T-Abs
  show ?case by iprover
next
  case (T-App  $t_1 T_{11} t_2 T_1 T_2$ )
  from  $\langle t_1 \cdot t_2 \in \text{value} \rangle$ 
  show ?case by cases
next
  case (T-TApp  $t_1 T_{11} T_{12} T_2 T_1 T_2'$ )
  from  $\langle t_1 \cdot_\tau T_2 \in \text{value} \rangle$ 
  show ?case by cases
next
  case (T-Sub  $t S T_1 T_2$ )
  from  $\langle \square \vdash S <: T_1 \rightarrow T_2 \rangle$ 
  obtain  $S_1 S_2$  where  $S: S = S_1 \rightarrow S_2$ 
  by cases (auto simp add: T-Sub)
  show ?case by (rule T-Sub S)+
next
  case (T-Let  $t_1 T_1 p \Delta t_2 T_2 T_1' T_2'$ )
  from  $\langle (\text{LET } p = t_1 \text{ IN } t_2) \in \text{value} \rangle$ 
  show ?case by cases
next
  case (T-Proj  $t fTs l T_1 T_2$ )
  from  $\langle t..l \in \text{value} \rangle$ 
  show ?case by cases
qed simp-all

lemma TyAll-canonical: — A.14(3)
  assumes ty:  $\square \vdash v : (\forall <: T_1. T_2)$ 
  shows  $v \in \text{value} \implies \exists t S. v = (\lambda <: S. t)$  using ty
proof (induct  $\square::\text{env } v : \forall <: T_1. T_2$  arbitrary:  $T_1 T_2$  rule: typing-induct)
  case (T-App  $t_1 T_{11} t_2 T_1 T_2$ )
  from  $\langle t_1 \cdot t_2 \in \text{value} \rangle$ 
  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  $\langle t_1 \cdot_\tau T_2 \in \text{value} \rangle$ 
  show ?case by cases
next

```

```

case ( $T\text{-Sub } t S T_1 T_2$ )
from  $\langle \square \vdash S <: (\forall <: T_1. T_2) \rangle$ 
obtain  $S_1 S_2$  where  $S: S = (\forall <: S_1. S_2)$ 
  by cases (auto simp add: T-Sub)
show ?case by (rule T-Sub S)+
next
case ( $T\text{-Let } t_1 T_1 p \Delta t_2 T_2 T_1' T_2'$ )
from  $\langle (LET\ p = t_1\ IN\ t_2) \in value \rangle$ 
show ?case by cases
next
case ( $T\text{-Proj } t fTs\ l\ T_1\ T_2$ )
from  $\langle t..l \in value \rangle$ 
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  $ty: \square \vdash v : RcdT\ fTs$ 
  shows  $v \in value \implies$ 
     $\exists fs. v = Rcd\ fs \wedge (\forall (l, t) \in set\ fs. t \in value)$  using  $ty$ 
proof (induct []::env v RcdT fTs arbitrary: fTs rule: typing-induct)
  case ( $T\text{-App } t_1 T_{11}\ t_2\ fTs$ )
    from  $\langle t_1 \cdot t_2 \in value \rangle$ 
    show ?case by cases
  next
  case ( $T\text{-TApp } t_1 T_{11}\ T_{12}\ T_2\ fTs$ )
    from  $\langle t_1 \cdot_\tau T_2 \in value \rangle$ 
    show ?case by cases
  next
  case ( $T\text{-Sub } t S\ fTs$ )
    from  $\langle \square \vdash S <: RcdT\ fTs \rangle$ 
    obtain  $fTs'$  where  $S: S = RcdT\ fTs'$ 
      by cases (auto simp add: T-Sub)
    show ?case by (rule T-Sub S)+
  next
  case ( $T\text{-Let } t_1 T_1 p \Delta t_2 T_2\ fTs$ )
    from  $\langle (LET\ p = t_1\ IN\ t_2) \in value \rangle$ 
    show ?case by cases
  next
  case ( $T\text{-Rcd } fs\ fTs$ )
    from  $\langle Rcd\ fs \in value \rangle$ 
    show ?case using  $T\text{-Rcd}$  by cases simp-all
  next
  case ( $T\text{-Proj } t fTs\ l\ fTs'$ )
    from  $\langle t..l \in value \rangle$ 
    show ?case by cases
qed simp-all

```

theorem *reorder-prop*:
 $\forall (l, t) \in \text{set fs}. P t \implies \forall (l, U) \in \text{set fTs}. \exists u. \text{fs}\langle l \rangle? = [u] \implies$
 $\forall (l, t) \in \text{set} (\text{map} (\lambda(l, T). (l, \text{the} (\text{fs}\langle l \rangle?))) \text{fTs}). P t$
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*:
 $\vdash p : T \Rightarrow \Delta \implies [] \vdash t : T \implies t \in \text{value} \implies$
 $\exists ts. \vdash p \triangleright t \Rightarrow ts$
 $\vdash \text{fps} [:] \text{fTs} \Rightarrow \Delta \implies [] \vdash \text{fs} [:] \text{fTs} \implies$
 $\forall (l, t) \in \text{set fs}. t \in \text{value} \implies \exists us. \vdash \text{fps} [\triangleright] \text{fs} \Rightarrow us$
proof (*induct arbitrary: t and fs set: ptyping ptyplings*)
case (*P-Var T t*)
show *?case* **by** (*iprover intro: M-PVar*)
next
case (*P-Rcd fps fTs Δ t*)
then obtain fs where
 $t : t = \text{Rcd fs}$ **and** $\text{fs} : \forall (l, t) \in \text{set fs}. t \in \text{value}$
by (*blast dest: RcdT-canonical*)
with P-Rcd have fs' : [] ⊢ Rcd fs : RcdT fTs **by** *simp*
hence [] ⊢ map (λ(l, T). (l, the (fs⟨l⟩?))) fTs [:] fTs
by (*rule Rcd-type2'*)
moreover from Rcd-type1' [OF fs']
have assoc : ∀ (l, U) ∈ set fTs. ∃ u. fs⟨l⟩? = [u] **by** *blast*
with fs have $\forall (l, t) \in \text{set} (\text{map} (\lambda(l, T). (l, \text{the} (\text{fs}\langle l \rangle?))) \text{fTs}). t \in \text{value}$
by (*rule reorder-prop*)
ultimately have $\exists us. \vdash \text{fps} [\triangleright] \text{map} (\lambda(l, T). (l, \text{the} (\text{fs}\langle l \rangle?))) \text{fTs} \Rightarrow us$
by (*rule P-Rcd*)
then obtain us where $\vdash \text{fps} [\triangleright] \text{map} (\lambda(l, T). (l, \text{the} (\text{fs}\langle l \rangle?))) \text{fTs} \Rightarrow us ..$
with P-Rcd(1) assoc have $\vdash \text{fps} [\triangleright] \text{fs} \Rightarrow us$ **by** (*rule matches-reorder'*)
hence $\vdash \text{PRcd fps} \triangleright \text{Rcd fs} \Rightarrow us$ **by** (*rule M-Rcd*)
with t show *?case* **by** *fastforce*
next

case ($P\text{-Nil } fs$)
show $?case$ **by** (*iprover intro: M-Nil*)
next
case ($P\text{-Cons } p \ T \ \Delta_1 \ fps \ fTs \ \Delta_2 \ l \ fs$)
from $\langle [] \vdash fs \ [:] \ (l, T) :: fTs \rangle$
obtain $t \ fs'$ **where** $fs: fs = (l, t) :: fs'$ **and** $t: [] \vdash t : T$
and $fs': [] \vdash fs' \ [:] \ fTs$ **by** *cases auto*
have $((l, t) :: fs') \langle l \rangle? = [t]$ **by** *simp*
moreover from $fs \ P\text{-Cons}$ **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 $P\text{-Cons } fs$ **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** $P\text{-Cons}(\delta)$ **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 $T\text{-Var}$
thus $?case$ **by** *simp*
next
case $T\text{-Abs}$
from $value.Abs$ **show** $?case$ **..**
next
case ($T\text{-App } t_1 \ T_{11} \ T_{12} \ t_2$)
hence $t_1 \in value \vee (\exists t'. t_1 \longmapsto t')$ **by** *simp*
thus $?case$
proof
assume $t_1\text{-val}: t_1 \in value$
with $T\text{-App}$ **obtain** $t \ S$ **where** $t_1: t_1 = (\lambda:S. t)$
by (*auto dest!: Fun-canonical*)
from $T\text{-App}$ **have** $t_2 \in value \vee (\exists t'. t_2 \longmapsto t')$ **by** *simp*
thus $?thesis$
proof
assume $t_2 \in value$
with t_1 **have** $t_1 \cdot t_2 \longmapsto t[0 \mapsto t_2]$
by *simp (rule eval-evals.intros)*
thus $?thesis$ **by** *iprover*
next
assume $\exists t'. t_2 \longmapsto t'$
then obtain t' **where** $t_2 \longmapsto t'$ **by** *iprover*
with $t_1\text{-val}$ **have** $t_1 \cdot t_2 \longmapsto t_1 \cdot t'$ **by** (*rule eval-evals.intros*)
thus $?thesis$ **by** *iprover*

```

    qed
  next
    assume  $\exists t'. t_1 \mapsto t'$ 
    then obtain  $t'$  where  $t_1 \mapsto t' ..$ 
    hence  $t_1 \cdot t_2 \mapsto t' \cdot t_2$  by (rule eval-vals.intros)
    thus ?thesis by iprover
  qed
next
  case T-TAbs
  from value.TAbs show ?case ..
next
  case (T-TApp t1 T11 T12 T2)
  hence  $t_1 \in \text{value} \vee (\exists t'. t_1 \mapsto t')$  by simp
  thus ?case
  proof
    assume  $t_1 \in \text{value}$ 
    with T-TApp obtain  $t S$  where  $t_1 = (\lambda <: S. t)$ 
    by (auto dest!: TyAll-canonical)
    hence  $t_1 \cdot_\tau T_2 \mapsto t[0 \mapsto_\tau T_2]$  by simp (rule eval-vals.intros)
    thus ?thesis by iprover
  next
    assume  $\exists t'. t_1 \mapsto t'$ 
    then obtain  $t'$  where  $t_1 \mapsto t' ..$ 
    hence  $t_1 \cdot_\tau T_2 \mapsto t' \cdot_\tau T_2$  by (rule eval-vals.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  $\Delta$  t2 T2)
  hence  $t_1 \in \text{value} \vee (\exists t'. t_1 \mapsto t')$  by simp
  thus ?case
  proof
    assume  $t_1: t_1 \in \text{value}$ 
    with T-Let have  $\exists ts. \vdash p \triangleright t_1 \Rightarrow ts$ 
    by (auto intro: ptyping-match)
    with  $t_1$  show ?thesis by (blast intro: eval-vals.intros)
  next
    assume  $\exists t'. t_1 \mapsto t'$ 
    thus ?thesis by (blast intro: eval-vals.intros)
  qed
next
  case (T-Rcd fs fTs)
  thus ?case by (blast intro: value.intros eval-vals.intros)
next
  case (T-Proj t fTs l T)
  hence  $t \in \text{value} \vee (\exists t'. t \mapsto t')$  by simp
  thus ?case

```

```

proof
  assume  $tv: t \in \text{value}$ 
  with  $T\text{-Proj}$  obtain  $fs$  where
     $t: t = \text{Rcd } fs$  and  $fs: \forall (l, t) \in \text{set } fs. t \in \text{value}$ 
    by ( $\text{auto dest: RcdT-canonical}$ )
  with  $T\text{-Proj}$  have  $\square \vdash \text{Rcd } fs : \text{RcdT } fTs$  by  $\text{simp}$ 
  hence  $\forall (l, U) \in \text{set } fTs. \exists u. fs\langle l \rangle? = \lfloor u \rfloor \wedge \square \vdash u : U$ 
    by ( $\text{rule Rcd-type1'}$ )
  with  $T\text{-Proj}$  obtain  $u$  where  $u: fs\langle l \rangle? = \lfloor u \rfloor$  by ( $\text{blast dest: assoc-set}$ )
  with  $fs$  have  $u \in \text{value}$  by ( $\text{blast dest: assoc-set}$ )
  with  $u \ t$  show  $?case$  by ( $\text{blast intro: eval-vals.intros}$ )
next
  assume  $\exists t'. t \mapsto t'$ 
  thus  $?case$  by ( $\text{blast intro: eval-vals.intros}$ )
qed
next
  case  $T\text{-Nil}$ 
  show  $?case$  by  $\text{simp}$ 
next
  case ( $T\text{-Cons } t \ T \ fs \ fTs \ l$ )
  thus  $?case$  by ( $\text{auto intro: eval-vals.intros}$ )
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

```

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

```

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

lemma *rctxt-labels*:

assumes $H: E \in \text{rctxt}$
shows $E t \langle l \rangle? = \perp \implies E t' \langle l \rangle? = \perp$ **using** H
by (*induct rule: rctxt-induct*) *auto*

The evaluation relation $t \mapsto_c t'$ 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 :: $\text{trm} \Rightarrow \text{trm} \Rightarrow \text{bool}$ (**infixl** \mapsto_c 50)

where

E-Ctxt: $t \mapsto_c t' \implies E \in \text{ctxt} \implies E t \mapsto_c E t'$
 | *E-Abs*: $v_2 \in \text{value} \implies (\lambda: T_{11}. t_{12}) \cdot v_2 \mapsto_c t_{12}[0 \mapsto v_2]$
 | *E-TAbs*: $(\lambda <: T_{11}. t_{12}) \cdot_\tau T_2 \mapsto_c t_{12}[0 \mapsto_\tau T_2]$
 | *E-LetV*: $v \in \text{value} \implies \vdash p \triangleright v \Rightarrow ts \implies (\text{LET } p = v \text{ IN } t) \mapsto_c t[0 \mapsto_s ts]$
 | *E-ProjRcd*: $fs \langle l \rangle? = [v] \implies v \in \text{value} \implies \text{Rcd } fs..l \mapsto_c v$

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

lemma *context-typing*: — A.18

$\Gamma \vdash u : T \implies E \in \text{ctxt} \implies u = E t \implies$
 $(\bigwedge T_0. \Gamma \vdash t : T_0 \implies \Gamma \vdash t' : T_0) \implies \Gamma \vdash E t' : T$
 $\Gamma \vdash fs [:] fTs \implies E_r \in \text{rctxt} \implies fs = E_r t \implies$
 $(\bigwedge T_0. \Gamma \vdash t : T_0 \implies \Gamma \vdash t' : T_0) \implies \Gamma \vdash E_r t' [:] fTs$

proof (*induct arbitrary: E t t' and E_r t t' set: typing typings*)

case (*T-Var* $\Gamma i U T E t t'$)
from $\langle E \in \text{ctxt} \rangle$
have $E = (\lambda t. t)$ **using** *T-Var* **by** *cases simp-all*
with *T-Var* **show** *?case* **by** (*blast intro: typing-typings.intros*)

next

case (*T-Abs* $T_1 T_2 \Gamma t_2 E t t'$)
from $\langle E \in \text{ctxt} \rangle$
have $E = (\lambda t. t)$ **using** *T-Abs* **by** *cases simp-all*
with *T-Abs* **show** *?case* **by** (*blast intro: typing-typings.intros*)

next

case (*T-App* $\Gamma t_1 T_{11} T_{12} t_2 E t t'$)
from $\langle E \in \text{ctxt} \rangle$
show *?case* **using** *T-App*
by *cases (simp-all, (blast intro: typing-typings.intros)+)*

next

case (*T-TAbs* $T_1 \Gamma t_2 T_2 E t t'$)

```

from  $\langle E \in \text{ctxt} \rangle$ 
have  $E = (\lambda t. t)$  using  $T\text{-TAbs}$  by cases  $\text{simp-all}$ 
with  $T\text{-TAbs}$  show  $?case$  by  $(\text{blast intro: typing-typings.intros})$ 
next
  case  $(T\text{-TApp } \Gamma \ t_1 \ T_{11} \ T_{12} \ T_2 \ E \ t \ t')$ 
  from  $\langle E \in \text{ctxt} \rangle$ 
  show  $?case$  using  $T\text{-TApp}$ 
    by cases  $(\text{simp-all}, (\text{blast intro: typing-typings.intros})+)$ 
next
  case  $(T\text{-Sub } \Gamma \ t \ S \ T \ E \ ta \ t')$ 
  thus  $?case$  by  $(\text{blast intro: typing-typings.intros})$ 
next
  case  $(T\text{-Let } \Gamma \ t_1 \ T_1 \ p \ \Delta \ t_2 \ T_2 \ E \ t \ t')$ 
  from  $\langle E \in \text{ctxt} \rangle$ 
  show  $?case$  using  $T\text{-Let}$ 
    by cases  $(\text{simp-all}, (\text{blast intro: typing-typings.intros})+)$ 
next
  case  $(T\text{-Rcd } \Gamma \ fs \ fTs \ E \ t \ t')$ 
  from  $\langle E \in \text{ctxt} \rangle$ 
  show  $?case$  using  $T\text{-Rcd}$ 
    by cases  $(\text{simp-all}, (\text{blast intro: typing-typings.intros})+)$ 
next
  case  $(T\text{-Proj } \Gamma \ t \ fTs \ l \ T \ E \ ta \ t')$ 
  from  $\langle E \in \text{ctxt} \rangle$ 
  show  $?case$  using  $T\text{-Proj}$ 
    by cases  $(\text{simp-all}, (\text{blast intro: typing-typings.intros})+)$ 
next
  case  $(T\text{-Nil } \Gamma \ E \ t \ t')$ 
  from  $\langle E \in \text{rctxt} \rangle$ 
  show  $?case$  using  $T\text{-Nil}$ 
    by cases  $\text{simp-all}$ 
next
  case  $(T\text{-Cons } \Gamma \ t \ T \ fs \ fTs \ l \ E \ ta \ t')$ 
  from  $\langle E \in \text{rctxt} \rangle$ 
  show  $?case$  using  $T\text{-Cons}$ 
    by cases  $(\text{blast intro: typing-typings.intros rctxt-labels})+$ 
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 $(T\text{-App } \Gamma \ T_{11} \ T_{12} \ t_2 \ T_{11}' \ t_{12})$

from $\langle \Gamma \vdash (\lambda: T_{11}'. t_{12}) : T_{11} \rightarrow T_{12} \rangle$

obtain S'

where $T_{11}: \Gamma \vdash T_{11} <: T_{11}'$
and $t_{12}: \text{VarB } T_{11}' :: \Gamma \vdash t_{12} : S'$
and $S': \Gamma \vdash S'[0 \mapsto_{\tau} \text{Top}]_{\tau} <: T_{12}$ **by** (rule *Abs-type'* [*simplified*]) *blast*
from $\langle \Gamma \vdash t_2 : T_{11} \rangle$
have $\Gamma \vdash t_2 : T_{11}'$ **using** T_{11} **by** (rule *T-Sub*)
with t_{12} **have** $\Gamma \vdash t_{12}[0 \mapsto t_2] : S'[0 \mapsto_{\tau} \text{Top}]_{\tau}$
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: T₁₁ t₁₂ T₂ rule: typing-induct*)
case (*T-TApp* $\Gamma T_{11} T_{12} T_2 T_{11}' t_{12}$)
from $\langle \Gamma \vdash (\lambda <: T_{11}'. t_{12}) : (\forall <: T_{11}. T_{12}) \rangle$
obtain S'
where $T\text{VarB } T_{11} :: \Gamma \vdash t_{12} : S'$
and $T\text{VarB } T_{11} :: \Gamma \vdash S' <: T_{12}$ **by** (rule *TAbs-type'*) *blast*
hence $T\text{VarB } T_{11} :: \Gamma \vdash t_{12} : T_{12}$ **by** (rule *T-Sub*)
then show *?case* **using** $\langle \Gamma \vdash T_2 <: T_{11} \rangle$
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 $H: \Gamma \vdash (\text{LET } p = t_1 \text{ IN } t_2) : T$
shows $p \triangleright t_1 \Rightarrow ts \Longrightarrow \Gamma \vdash t_2[0 \mapsto_s ts] : T$
using H
proof (*induct* $\Gamma \text{LET } p = t_1 \text{ IN } t_2 T$ *arbitrary: p t₁ t₂ ts rule: typing-induct*)
case (*T-Let* $\Gamma t_1 T_1 p \Delta t_2 T_2 ts$)
from $\langle \vdash p : T_1 \Rightarrow \Delta \rangle \langle \Gamma \vdash t_1 : T_1 \rangle \langle \Delta @ \Gamma \vdash t_2 : T_2 \rangle \langle \vdash p \triangleright t_1 \Rightarrow ts \rangle$
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 $H: \Gamma \vdash \text{Rcd } fs..l : T$
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  $\langle \Gamma \vdash \text{Rcd fs} : \text{RcdT fTs} \rangle$ 
  have  $\forall (l, U) \in \text{set fTs}. \exists u. \text{fs}\langle l \rangle? = [u] \wedge \Gamma \vdash u : U$ 
    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  $H: t \mapsto_c t'$ 
  shows  $\Gamma \vdash t : T \implies \Gamma \vdash t' : T$  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 v2 T11 t12  $\Gamma$  T)
  from E-Abs(2)
  show ?case by (rule Abs-preservation)
next
  case (E-TAbs T11 t12 T2  $\Gamma$  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
   $\llbracket \vdash t : T \implies$ 
     $t \in \text{value} \vee (\exists E t_0 t_0'. E \in \text{ctxt} \wedge t = E t_0 \wedge t_0 \mapsto_c t_0')$ 
   $\llbracket \vdash \text{fs } [:] \text{ fTs} \implies$ 
     $(\forall (l, t) \in \text{set fs}. t \in \text{value}) \vee (\exists E t_0 t_0'. E \in \text{rcctx} \wedge \text{fs} = E t_0 \wedge t_0 \mapsto_c t_0')$ 
proof (induct  $\llbracket :: \text{env } t \text{ T}$  and  $\llbracket :: \text{env fs fTs set: typing typings}$ )
  case T-Var
  thus ?case by simp

```

```

next
  case T-Abs
  from value.Abs show ?case ..
next
  case (T-App t1 T11 T12 t2)
  from ⟨t1 ∈ value ∨ (∃ E t0 t0'. E ∈ ctxt ∧ t1 = E t0 ∧ t0 ↦c t0')⟩
  show ?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 ⟨t2 ∈ value ∨ (∃ E t0 t0'. E ∈ ctxt ∧ t2 = E t0 ∧ t0 ↦c t0')⟩
    show ?thesis
  proof
    assume t2 ∈ value
    with t1 have t1 · t2 ↦c t[0 ↦ t2]
    by simp (rule eval.intros)
    thus ?thesis by (iprover intro: C-Hole)
  next
    assume ∃ E t0 t0'. E ∈ ctxt ∧ t2 = E t0 ∧ t0 ↦c t0'
    with t1-val show ?thesis by (iprover intro: ctxt-rctx.intros)
  qed
  next
    assume ∃ E t0 t0'. E ∈ ctxt ∧ t1 = E t0 ∧ t0 ↦c t0'
    thus ?thesis by (iprover intro: ctxt-rctx.intros)
  qed
next
  case T-TAbs
  from value.TAbs show ?case ..
next
  case (T-TApp t1 T11 T12 T2)
  from ⟨t1 ∈ value ∨ (∃ E t0 t0'. E ∈ ctxt ∧ t1 = E t0 ∧ t0 ↦c t0')⟩
  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-rctx.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  $t_1: t_1 \in \text{value}$ 
  with T-Let have  $\exists ts. \vdash p \triangleright t_1 \Rightarrow ts$ 
  by (auto intro: ptying-match)
  with  $t_1$  show ?thesis by (iprover intro: eval.intros C-Hole)
next
  assume  $\exists E t_0 t_0'. E \in \text{ctxt} \wedge t_1 = E t_0 \wedge t_0 \mapsto_c t_0'$ 
  thus ?thesis by (iprover intro: ctxt-rctxt.intros)
qed
next
  case (T-Rcd fs fTs)
  thus ?case by (blast intro: value.intros eval.intros ctxt-rctxt.intros)
next
  case (T-Proj t fTs l T)
  from  $\langle t \in \text{value} \vee (\exists E t_0 t_0'. E \in \text{ctxt} \wedge t = E t_0 \wedge t_0 \mapsto_c t_0') \rangle$ 
  show ?case
  proof
    assume  $tv: t \in \text{value}$ 
    with T-Proj obtain fs where
       $t: t = \text{Rcd fs}$  and  $fs: \forall (l, t) \in \text{set fs}. t \in \text{value}$ 
    by (auto dest: RcdT-canonical)
    with T-Proj have  $\square \vdash \text{Rcd fs} : \text{RcdT fTs}$  by simp
    hence  $\forall (l, U) \in \text{set fTs}. \exists u. fs \langle l \rangle? = \lfloor u \rfloor \wedge \square \vdash u : U$ 
    by (rule Rcd-type1')
    with T-Proj obtain  $u$  where  $u: fs \langle l \rangle? = \lfloor u \rfloor$  by (blast dest: assoc-set)
    with fs have  $u \in \text{value}$  by (blast dest: assoc-set)
    with  $u t$  show ?thesis by (iprover intro: eval.intros C-Hole)
  next
    assume  $\exists E t_0 t_0'. E \in \text{ctxt} \wedge t = E t_0 \wedge t_0 \mapsto_c t_0'$ 
    thus ?case by (iprover intro: ctxt-rctxt.intros)
  qed
next
  case T-Nil
  show ?case by simp
next
  case (T-Cons t T fs fTs l)
  thus ?case by (auto intro: ctxt-rctxt.intros)
qed

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

```

Finally, we prove that the two definitions of the evaluation relation are

equivalent. The proof that $t \mapsto_c t'$ implies $t \mapsto t'$ requires a lemma stating that \mapsto is compatible with evaluation contexts.

lemma *ctxt-imp-eval*:

$E \in \text{ctxt} \implies t \mapsto t' \implies E t \mapsto E t'$
 $E_r \in \text{rctxt} \implies t \mapsto t' \implies E_r t [\mapsto] E_r t'$
by (*induct rule: ctxt-rctxt.inducts*) (*auto intro: eval-vals.intros*)

lemma *eval-evalc-eq*: $(t \mapsto t') = (t \mapsto_c t')$

proof

fix $ts\ ts'$

have $r: t \mapsto t' \implies t \mapsto_c t'$ **and**

$ts [\mapsto] ts' \implies \exists E t t'. E \in \text{rctxt} \wedge ts = E t \wedge ts' = E t' \wedge t \mapsto_c t'$

by (*induct rule: eval-vals.inducts*) (*iprover intro: ctxt-rctxt.intros eval.intros*)+

assume $t \mapsto t'$

thus $t \mapsto_c t'$ **by** (*rule r*)

next

assume $t \mapsto_c t'$

thus $t \mapsto t'$

by *induct* (*auto intro: eval-vals.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 \mapsto , we introduce the inductive predicate $t \Downarrow u$, denoting that u is a normal form of t .

inductive *norm* :: $\text{trm} \Rightarrow \text{trm} \Rightarrow \text{bool}$ (**infixl** \Downarrow 50)

where

$t \in \text{value} \implies t \Downarrow t$

$| t \mapsto s \implies s \Downarrow u \implies t \Downarrow u$

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* $t \implies \text{valuep}$ (*Rcd fs*) $\implies \text{valuep}$ (*Rcd ((l, t) # fs)*)

```

by (auto intro!: valuep.intros elim!: valuep.cases)

lemmas valuep.intros(1)[code-pred-intro Abs1] valuep.intros(2)[code-pred-intro TAbs1]

code-pred (modes: i => bool) valuep
proof -
  case valuep
  from valuep.premis 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.Abs1 show thesis .
  next
  case TAbs
  with valuep.TAbs1 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 ∈ value ↔ 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)),

```

$$\begin{aligned}
& ("multpp", PVar (natT \to natT \to natT))] = Rcd \\
& [("multpp", \lambda:natT. \lambda:natT. \lambda<:Top. \lambda<:TVar 0. \lambda<:TVar 1. \lambda:TVar 2 \to \\
& TVar 1. \\
& \quad Var 5 \cdot_{\tau} TVar 3 \cdot_{\tau} TVar 2 \cdot_{\tau} TVar 1 \cdot (Var 4 \cdot_{\tau} TVar 3 \cdot_{\tau} TVar 2 \cdot_{\tau} \\
& TVar 1) \cdot Var 0), \\
& ("pluspp", \lambda:natT. \lambda:natT. \lambda<:Top. \lambda<:TVar 0. \lambda<:TVar 1. \lambda:TVar 2 \to \\
& TVar 1. \lambda:TVar 1. \\
& \quad Var 6 \cdot_{\tau} TVar 4 \cdot_{\tau} TVar 3 \cdot_{\tau} TVar 3 \cdot Var 1 \cdot \\
& \quad (Var 5 \cdot_{\tau} TVar 4 \cdot_{\tau} TVar 3 \cdot_{\tau} TVar 2 \cdot Var 1 \cdot Var 0))] \\
& IN \\
& \quad Var 0 \cdot (Var 1 \cdot Var 2 \cdot Var 2) \cdot Var 2
\end{aligned}$$

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

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.