A Solution to the PoplMark Challenge in Isabelle/HOL

Stefan Berghofer

March 17, 2025

Abstract

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

Contents

1 General Utilities

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

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

lemma *True-simps*:
$(\mathit{True} \Longrightarrow \mathit{PROP}\ P) \equiv \mathit{PROP}\ P$
$(\mathit{PROP}\ P \Longrightarrow \mathit{True}) \equiv \mathit{PROP}\ \mathit{Trueprop}\ \mathit{True}$
$(\bigwedge x.\ \mathit{True}) \equiv \mathit{PROP}\ \mathit{Trueprop}\ \mathit{True}$
⟨*proof*⟩

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

lemma *ballpI*: $(\bigwedge x\ y.\ (x,\ y) \in A \Longrightarrow P\ x\ y) \Longrightarrow \forall (x,\ y) \in A.\ P\ x\ y$
⟨*proof*⟩

lemma *bpspec*: $\forall (x,\ y) \in A.\ P\ x\ y \Longrightarrow (x,\ y) \in A \Longrightarrow P\ x\ y$
⟨*proof*⟩

lemma *ballpE*: $\forall (x,\ y) \in A.\ P\ x\ y \Longrightarrow (P\ x\ y \Longrightarrow Q) \Longrightarrow$
$((x,\ y) \notin A \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *bexpI*: $P\ x\ y \Longrightarrow (x,\ y) \in A \Longrightarrow \exists (x,\ y) \in A.\ P\ x\ y$
⟨*proof*⟩

lemma *bexpE*: $\exists (x,\ y) \in A.\ P\ x\ y \Longrightarrow$
$(\bigwedge x\ y.\ (x,\ y) \in A \Longrightarrow P\ x\ y \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

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

lemma *wf-measure-size*: *wf* (*measure size*) ⟨*proof*⟩

notation
Some (‹⌊-⌋›)

notation
None (‹⊥›)

notation
length (‹‖-‖›)

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

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

primrec
$nth\text{-}el :: 'a\ list \Rightarrow nat \Rightarrow 'a\ option$ (‹-⟨-⟩› [90, 0] 91)
where
$[]\langle i\rangle = \bot$
$|\ (x\ \#\ xs)\langle i\rangle = (case\ i\ of\ 0 \Rightarrow \lfloor x\rfloor\ |\ Suc\ j \Rightarrow xs\ \langle j\rangle)$

lemma *nth-el-append1* [*simp*]: $i < \|xs\| \Longrightarrow (xs\ @\ ys)\langle i\rangle = xs\langle i\rangle$
⟨*proof*⟩

lemma *nth-el-append2* [*simp*]: $\|xs\| \leq i \Longrightarrow (xs\ @\ ys)\langle i\rangle = ys\langle i - \|xs\|\rangle$
⟨*proof*⟩

Association lists

primrec $assoc :: ('a \times 'b)\ list \Rightarrow 'a \Rightarrow 'b\ option$ (‹-⟨-⟩$_?$› [90, 0] 91)
where
$[]\langle a\rangle_? = \bot$
$|\ (x\ \#\ xs)\langle a\rangle_? = (if\ fst\ x = a\ then\ \lfloor snd\ x\rfloor\ else\ xs\langle a\rangle_?)$

primrec $unique :: ('a \times 'b)\ list \Rightarrow bool$
where
$unique\ [] = True$
$|\ unique\ (x\ \#\ xs) = (xs\langle fst\ x\rangle_? = \bot \wedge unique\ xs)$

lemma *assoc-set*: $ps\langle x\rangle_? = \lfloor y\rfloor \Longrightarrow (x, y) \in set\ ps$
⟨*proof*⟩

lemma *map-assoc-None* [*simp*]:
$ps\langle x\rangle_? = \bot \Longrightarrow map\ (\lambda(x, y).\ (x, f\ x\ y))\ ps\langle x\rangle_? = \bot$
⟨*proof*⟩

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

2 Formalization of the basic calculus

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

2.1 Types and Terms

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

datatype *type* =
TVar nat

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

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

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

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

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

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

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

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

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

2.2 Lifting and Substitution

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

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

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

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

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

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

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

| $(\lambda{:}T.\ t)[k \mapsto_\tau S] = (\lambda{:}T[k \mapsto_\tau S]_\tau.\ t[k{+}1 \mapsto_\tau S])$
| $(\lambda{<:}T.\ t)[k \mapsto_\tau S] = (\lambda{<:}T[k \mapsto_\tau S]_\tau.\ t[k{+}1 \mapsto_\tau S])$

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-length* [*simp*]: $\|\uparrow_e\ n\ k\ \Gamma\| = \|\Gamma\|$
⟨*proof*⟩

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

lemma *liftE-nth* [*simp*]:
$(\uparrow_e\ n\ k\ \Gamma)\langle i\rangle = map\text{-}option\ (mapB\ (\uparrow_\tau\ n\ (k + \|\Gamma\| - i - 1)))\ (\Gamma\langle i\rangle)$
⟨*proof*⟩

lemma *substE-nth* [*simp*]:
$(\Gamma[0 \mapsto_\tau T]_e)\langle i\rangle = map\text{-}option\ (mapB\ (\lambda U.\ U[\|\Gamma\| - i - 1 \mapsto_\tau T]_\tau))\ (\Gamma\langle i\rangle)$
⟨*proof*⟩

lemma *liftT-liftT* [*simp*]:
$i \le j \Longrightarrow j \le i + m \Longrightarrow \uparrow_\tau\ n\ j\ (\uparrow_\tau\ m\ i\ T) = \uparrow_\tau\ (m + n)\ i\ T$
⟨*proof*⟩

lemma *liftT-liftT'* [*simp*]:
$i + m \le j \Longrightarrow \uparrow_\tau\ n\ j\ (\uparrow_\tau\ m\ i\ T) = \uparrow_\tau\ m\ i\ (\uparrow_\tau\ n\ (j - m)\ T)$

$\langle proof \rangle$

lemma *lift-size* [*simp*]: $size\ (\uparrow_\tau\ n\ k\ T) = size\ T$
$\langle proof \rangle$

lemma *liftT0* [*simp*]: $\uparrow_\tau\ 0\ i\ T = T$
$\langle proof \rangle$

lemma *lift0* [*simp*]: $\uparrow\ 0\ i\ t = t$
$\langle proof \rangle$

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

theorem *liftT-substT* [*simp*]:
$k \le k' \Longrightarrow \uparrow_\tau\ n\ k\ (T[k' \mapsto_\tau U]_\tau) = \uparrow_\tau\ n\ k\ T[k' + n \mapsto_\tau U]_\tau$
$\langle proof \rangle$

theorem *liftT-substT'* [*simp*]: $k' < k \Longrightarrow$
$\uparrow_\tau\ n\ k\ (T[k' \mapsto_\tau U]_\tau) = \uparrow_\tau\ n\ (k + 1)\ T[k' \mapsto_\tau \uparrow_\tau\ n\ (k - k')\ U]_\tau$
$\langle proof \rangle$

lemma *liftT-substT-Top* [*simp*]:
$k \le k' \Longrightarrow \uparrow_\tau\ n\ k'\ (T[k \mapsto_\tau Top]_\tau) = \uparrow_\tau\ n\ (Suc\ k')\ T[k \mapsto_\tau Top]_\tau$
$\langle proof \rangle$

lemma *liftT-substT-strange*:
$\uparrow_\tau\ n\ k\ T[n + k \mapsto_\tau U]_\tau = \uparrow_\tau\ n\ (Suc\ k)\ T[k \mapsto_\tau \uparrow_\tau\ n\ 0\ U]_\tau$
$\langle proof \rangle$

lemma *lift-lift* [*simp*]:
$k \le k' \Longrightarrow k' \le k + n \Longrightarrow \uparrow\ n'\ k'\ (\uparrow\ n\ k\ t) = \uparrow\ (n + n')\ k\ t$
$\langle proof \rangle$

lemma *substT-substT*:
$i \le j \Longrightarrow T[Suc\ j \mapsto_\tau V]_\tau[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_\tau = T[i \mapsto_\tau U]_\tau[j \mapsto_\tau V]_\tau$
$\langle proof \rangle$

2.3 Well-formedness

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

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

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

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

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

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

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

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

lemma *wf-TVarB*: $\Gamma \vdash_{wf}\ T \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow TVarB\ T :: \Gamma \vdash_{wf}$
  ⟨*proof*⟩

lemma *wf-VarB*: $\Gamma \vdash_{wf}\ T \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow VarB\ T :: \Gamma \vdash_{wf}$
  ⟨*proof*⟩

lemma *map-is-TVarb*:
  $map\ is\text{-}TVarB\ \Gamma' = map\ is\text{-}TVarB\ \Gamma \Longrightarrow$
    $\Gamma\langle i\rangle = \lfloor TVarB\ T\rfloor \Longrightarrow \exists\, T.\ \Gamma'\langle i\rangle = \lfloor TVarB\ T\rfloor$
⟨*proof*⟩

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 $map\ is\text{-}TVarB\ \Gamma' = map\ is\text{-}TVarB\ \Gamma \Longrightarrow \Gamma' \vdash_{wf}\ T$ ⟨*proof*⟩

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

lemma *wfE-replace*:
$\Delta \ @ \ B :: \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf B} \ B' \Longrightarrow \textit{is-TVarB} \ B' = \textit{is-TVarB} \ B \Longrightarrow$
$\Delta \ @ \ B' :: \Gamma \vdash_{wf}$
⟨*proof*⟩

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 $\uparrow_e \ (\textit{Suc 0}) \ \textit{0} \ \Delta \ @ \ B :: \Gamma \vdash_{wf} \uparrow_\tau \ (\textit{Suc 0}) \ \|\Delta\| \ T$
⟨*proof*⟩

lemma *wf-weaken'*: $\Gamma \vdash_{wf} \ T \Longrightarrow \Delta \ @ \ \Gamma \vdash_{wf} \uparrow_\tau \ \|\Delta\| \ \textit{0} \ T$
⟨*proof*⟩

lemma *wfE-weaken*: $\Delta \ @ \ \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf B} \ B \Longrightarrow \uparrow_e \ (\textit{Suc 0}) \ \textit{0} \ \Delta \ @ \ B :: \Gamma \vdash_{wf}$
⟨*proof*⟩

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

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

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

theorem *wf-subst*:
$\Delta \ @ \ B :: \Gamma \vdash_{wf} \ T \Longrightarrow \Gamma \vdash_{wf} \ U \Longrightarrow \Delta[\textit{0} \mapsto_\tau \ U]_e \ @ \ \Gamma \vdash_{wf} \ T[\|\Delta\| \mapsto_\tau \ U]_\tau$
⟨*proof*⟩

theorem *wfE-subst*: $\Delta \ @ \ B :: \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} \ U \Longrightarrow \Delta[\textit{0} \mapsto_\tau \ U]_e \ @ \ \Gamma \vdash_{wf}$
⟨*proof*⟩

2.4 Subtyping

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

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

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

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

lemma *wf-subtype*:
  assumes *PQ*: $\Gamma \vdash P <: Q$
  shows $\Gamma \vdash_{wf} P \wedge \Gamma \vdash_{wf} Q$ ⟨*proof*⟩

lemma *wf-subtypeE*:
  assumes *H*: $\Gamma \vdash T <: U$
  and *H'*: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow P$
  shows *P*
  ⟨*proof*⟩

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

lemma *wf-subtype-conj*:
  $\Gamma \vdash T <: U \Longrightarrow \Gamma \vdash_{wf} \wedge \Gamma \vdash_{wf} T \wedge \Gamma \vdash_{wf} U$
  ⟨*proof*⟩

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

lemma *subtype-refl*: — A.1
  $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash T <: T$
  ⟨*proof*⟩

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: Δ @ $\Gamma \vdash P <: Q$
  and wf: $\Gamma \vdash_{wf B} B$
  shows $\uparrow_e$ *1 0* Δ @ $B :: \Gamma \vdash \uparrow_\tau$ *1* $\|\Delta\|$ $P <: \uparrow_\tau$ *1* $\|\Delta\|$ Q ⟨*proof*⟩

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

lemma *subtype-weaken'*: — A.2
  $\Gamma \vdash P <: Q \Longrightarrow \Delta$ @ $\Gamma \vdash_{wf} \Longrightarrow \Delta$ @ $\Gamma \vdash \uparrow_\tau \|\Delta\|$ *0* $P <: \uparrow_\tau \|\Delta\|$ *0* Q
⟨*proof*⟩

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

lemma *subtype-trans*: — A.3
  $\Gamma \vdash S <: Q \Longrightarrow \Gamma \vdash Q <: T \Longrightarrow \Gamma \vdash S <: T$
  Δ @ *TVarB* $Q :: \Gamma \vdash M <: N \Longrightarrow \Gamma \vdash P <: Q \Longrightarrow$
    Δ @ *TVarB* $P :: \Gamma \vdash M <: N$
  ⟨*proof*⟩

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

lemma *substT-subtype*: — A.10
  assumes H: Δ @ *TVarB* $Q :: \Gamma \vdash S <: T$

shows $\Gamma \vdash P <: Q \Longrightarrow \Delta[\mathit{0} \mapsto_\tau P]_e$ @ $\Gamma \vdash S[\|\Delta\| \mapsto_\tau P]_\tau <: T[\|\Delta\| \mapsto_\tau P]_\tau$
⟨*proof*⟩

lemma *subst-subtype*:
assumes H: Δ @ *VarB* V :: $\Gamma \vdash T <: U$
shows $\downarrow_e \mathit{1}\ \mathit{0}\ \Delta$ @ $\Gamma \vdash \downarrow_\tau \mathit{1}\ \|\Delta\|\ T <: \downarrow_\tau \mathit{1}\ \|\Delta\|\ U$
⟨*proof*⟩

2.5 Typing

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

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

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

theorem *wf-typeE1*:
assumes H: $\Gamma \vdash t : T$
shows $\Gamma \vdash_{wf}$ ⟨*proof*⟩

theorem *wf-typeE2*:
assumes H: $\Gamma \vdash t : T$
shows $\Gamma \vdash_{wf} T$ ⟨*proof*⟩

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

lemma *wf-type-conj*: $\Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wf} \wedge \Gamma \vdash_{wf} T$
⟨*proof*⟩

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

lemma *narrow-type*: — A.7
assumes H: Δ @ *TVarB* Q :: $\Gamma \vdash t : T$
shows $\Gamma \vdash P <: Q \Longrightarrow \Delta$ @ *TVarB* P :: $\Gamma \vdash t : T$

$\langle proof \rangle$

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

lemma *Abs-type*: — A.13(1)
assumes H: $\Gamma \vdash (\lambda{:}S.\ s) : T$
shows $\Gamma \vdash T <: U \rightarrow U' \Longrightarrow$
$(\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow \mathit{VarB}\ S :: \Gamma \vdash s : S' \Longrightarrow$
$\Gamma \vdash \downarrow_\tau\ 1\ 0\ S' <: U' \Longrightarrow P) \Longrightarrow P$
$\langle proof \rangle$

lemma *Abs-type'*:
assumes H: $\Gamma \vdash (\lambda{:}S.\ s) : U \rightarrow U'$
and R: $\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow \mathit{VarB}\ S :: \Gamma \vdash s : S' \Longrightarrow$
$\Gamma \vdash \downarrow_\tau\ 1\ 0\ S' <: U' \Longrightarrow P$
shows P
$\langle proof \rangle$

lemma *TAbs-type*: — A.13(2)
assumes H: $\Gamma \vdash (\lambda{<:}S.\ s) : T$
shows $\Gamma \vdash T <: (\forall{<:}U.\ U') \Longrightarrow$
$(\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow \mathit{TVarB}\ U :: \Gamma \vdash s : S' \Longrightarrow$
$\mathit{TVarB}\ U :: \Gamma \vdash S' <: U' \Longrightarrow P) \Longrightarrow P$
$\langle proof \rangle$

lemma *TAbs-type'*:
assumes H: $\Gamma \vdash (\lambda{<:}S.\ s) : (\forall{<:}U.\ U')$
and R: $\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow \mathit{TVarB}\ U :: \Gamma \vdash s : S' \Longrightarrow$
$\mathit{TVarB}\ U :: \Gamma \vdash S' <: U' \Longrightarrow P$
shows P $\langle proof \rangle$

lemma *T-eq*: $\Gamma \vdash t : T \Longrightarrow T = T' \Longrightarrow \Gamma \vdash t : T'$ $\langle proof \rangle$

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

lemma *type-weaken*:
assumes H: $\Delta\ @\ \Gamma \vdash t : T$
shows $\Gamma \vdash_{wf B} B \Longrightarrow$
$\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma \vdash \uparrow\ 1\ \|\Delta\|\ t : \uparrow_\tau\ 1\ \|\Delta\|\ T$ $\langle proof \rangle$

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)
$\Gamma \vdash t : T \Longrightarrow \Delta\ @\ \Gamma \vdash_{wf} \Longrightarrow \Delta\ @\ \Gamma \vdash \uparrow\ \|\Delta\|\ 0\ t : \uparrow_\tau\ \|\Delta\|\ 0\ T$
$\langle proof \rangle$

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: Δ @ *VarB* U :: $\Gamma \vdash t : T$
  shows $\Gamma \vdash u : U \Longrightarrow$
    $\downarrow_e$ *1 0* Δ @ $\Gamma \vdash$ (*subst* t (*length* Δ) u) : $\downarrow_\tau$ *1* $\|\Delta\|$ T ⟨*proof*⟩

theorem *substT-type*: — A.11
  assumes H: Δ @ *TVarB* Q :: $\Gamma \vdash t : T$
  shows $\Gamma \vdash P <: Q \Longrightarrow$
    $\Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash t[\|\Delta\| \mapsto_\tau P] : T[\|\Delta\| \mapsto_\tau P]_\tau$ ⟨*proof*⟩

2.6 Evaluation

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

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

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

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

| *E-TApp*: $t \longmapsto t' \Longrightarrow t \cdot_\tau T \longmapsto t' \cdot_\tau T$

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

theorem *preservation*: — A.20
  assumes H: $\Gamma \vdash t : T$
  shows $t \longmapsto t' \Longrightarrow \Gamma \vdash t' : T$ ⟨*proof*⟩

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

lemma *Fun-canonical*: — A.14(1)
  assumes *ty*: $[] \vdash v : T_1 \rightarrow T_2$
  shows $v \in value \Longrightarrow \exists t\ S.\ v = (\lambda{:}S.\ t)$ ⟨*proof*⟩

lemma *TyAll-canonical*: — A.14(3)
  assumes *ty*: $[] \vdash v : (\forall <:T_1.\ T_2)$
  shows $v \in value \Longrightarrow \exists t\ S.\ v = (\lambda <:S.\ t)$ ⟨*proof*⟩

theorem *progress*:
  assumes *ty*: $[] \vdash t : T$
  shows $t \in value \vee (\exists t'.\ t \longmapsto t')$ ⟨*proof*⟩

3 Extending the calculus with records

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

3.1 Types and Terms

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

type-synonym *name* = *string*

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

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

datatype *binding* = *VarB type* | *TVarB type*

type-synonym *env* = *binding list*

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

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

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

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

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

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

| *Proj trm name* (‹(-..-)› [*90*, *91*] *90*)
| *LET pat trm trm* (‹(*LET* (- =/ -)/ *IN* (-))› *10*)

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

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

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

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

3.2 Lifting and Substitution

primrec *psize* :: *pat* ⇒ *nat* (‹$\|$-$\|_p$›)
  and *rsize* :: *rpat* ⇒ *nat* (‹$\|$-$\|_r$›)
  and *fsize* :: *fpat* ⇒ *nat* (‹$\|$-$\|_f$›)
where
  $\|PVar\ T\|_p = 1$
| $\|PRcd\ fs\|_p = \|fs\|_r$
| $\|[]\|_r = 0$
| $\|f :: fs\|_r = \|f\|_f + \|fs\|_r$
| $\|(l,\ p)\|_f = \|p\|_p$

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

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

primrec *lift* :: *nat* ⇒ *nat* ⇒ *trm* ⇒ *trm* (‹↑›)
  and *liftr* :: *nat* ⇒ *nat* ⇒ *rcd* ⇒ *rcd* (‹$\uparrow_r$›)
  and *liftf* :: *nat* ⇒ *nat* ⇒ *fld* ⇒ *fld* (‹$\uparrow_f$›)
where

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

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

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

primrec *substpT* :: *pat* ⇒ *nat* ⇒ *type* ⇒ *pat* (‹-[- $\mapsto_\tau$ -]$_p$› [300, 0, 0] 300)
  and *substrpT* :: *rpat* ⇒ *nat* ⇒ *type* ⇒ *rpat* (‹-[- $\mapsto_\tau$ -]$_{rp}$› [300, 0, 0] 300)
  and *substfpT* :: *fpat* ⇒ *nat* ⇒ *type* ⇒ *fpat* (‹-[- $\mapsto_\tau$ -]$_{fp}$› [300, 0, 0] 300)
where

$$
\begin{array}{l}
\ \ (PVar\ T)[k \mapsto_\tau S]_p = PVar\ (T[k \mapsto_\tau S]_\tau) \\
|\ (PRcd\ fs)[k \mapsto_\tau S]_p = PRcd\ (fs[k \mapsto_\tau S]_{rp}) \\
|\ [][k \mapsto_\tau S]_{rp} = [] \\
|\ (f :: fs)[k \mapsto_\tau S]_{rp} = f[k \mapsto_\tau S]_{fp} :: fs[k \mapsto_\tau S]_{rp} \\
|\ (l,\ p)[k \mapsto_\tau S]_{fp} = (l,\ p[k \mapsto_\tau S]_p)
\end{array}
$$

primrec *decp* :: *nat* ⇒ *nat* ⇒ *pat* ⇒ *pat* (‹$\downarrow_p$›)
where

$$
\begin{array}{l}
\ \ \downarrow_p\ 0\ k\ p = p \\
|\ \downarrow_p\ (Suc\ n)\ k\ p = \downarrow_p\ n\ k\ (p[k \mapsto_\tau Top]_p)
\end{array}
$$

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

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

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

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

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

primrec *substE* :: *env* ⇒ *nat* ⇒ *type* ⇒ *env* (‹-[- ↦$_\tau$ -]$_e$› [*300*, *0*, *0*] *300*)

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

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

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

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

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

primrec *decrT* :: *nat* ⇒ *nat* ⇒ *rcdT* ⇒ *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*]: $\|\uparrow_e\ n\ k\ \Gamma\| = \|\Gamma\|$
  ⟨*proof*⟩

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

lemma *liftE-nth* [*simp*]:
  $(\uparrow_e\ n\ k\ \Gamma)\langle i\rangle = map\text{-}option\ (mapB\ (\uparrow_\tau\ n\ (k + \|\Gamma\| - i - 1)))\ (\Gamma\langle i\rangle)$
  ⟨*proof*⟩

lemma *substE-nth* [*simp*]:
  $(\Gamma[0 \mapsto_\tau T]_e)\langle i\rangle = map\text{-}option\ (mapB\ (\lambda U.\ U[\|\Gamma\| - i - 1 \mapsto_\tau T]_\tau))\ (\Gamma\langle i\rangle)$
  ⟨*proof*⟩

lemma *liftT-liftT* [*simp*]:
  $i \le j \Longrightarrow j \le i + m \Longrightarrow \uparrow_\tau\ n\ j\ (\uparrow_\tau\ m\ i\ T) = \uparrow_\tau\ (m + n)\ i\ T$
  $i \le j \Longrightarrow j \le i + m \Longrightarrow \uparrow_{r\tau}\ n\ j\ (\uparrow_{r\tau}\ m\ i\ rT) = \uparrow_{r\tau}\ (m + n)\ i\ rT$
  $i \le j \Longrightarrow j \le i + m \Longrightarrow \uparrow_{f\tau}\ n\ j\ (\uparrow_{f\tau}\ m\ i\ fT) = \uparrow_{f\tau}\ (m + n)\ i\ fT$
  ⟨*proof*⟩

lemma *liftT-liftT'* [*simp*]:
$i + m \leq j \Longrightarrow \uparrow_\tau\ n\ j\ (\uparrow_\tau\ m\ i\ T) = \uparrow_\tau\ m\ i\ (\uparrow_\tau\ n\ (j - m)\ T)$
$i + m \leq j \Longrightarrow \uparrow_{r\tau}\ n\ j\ (\uparrow_{r\tau}\ m\ i\ rT) = \uparrow_{r\tau}\ m\ i\ (\uparrow_{r\tau}\ n\ (j - m)\ rT)$
$i + m \leq j \Longrightarrow \uparrow_{f\tau}\ n\ j\ (\uparrow_{f\tau}\ m\ i\ fT) = \uparrow_{f\tau}\ m\ i\ (\uparrow_{f\tau}\ n\ (j - m)\ fT)$
⟨*proof*⟩

lemma *lift-size* [*simp*]:
$size\ (\uparrow_\tau\ n\ k\ T) = size\ T$
$size\text{-}list\ (size\text{-}prod\ (\lambda x.\ 0)\ size)\ (\uparrow_{r\tau}\ n\ k\ rT) = size\text{-}list\ (size\text{-}prod\ (\lambda x.\ 0)\ size)$
rT
$size\text{-}prod\ (\lambda x.\ 0)\ size\ (\uparrow_{f\tau}\ n\ k\ fT) = size\text{-}prod\ (\lambda x.\ 0)\ size\ fT$
⟨*proof*⟩

lemma *liftT0* [*simp*]:
$\uparrow_\tau\ 0\ i\ T = T$
$\uparrow_{r\tau}\ 0\ i\ rT = rT$
$\uparrow_{f\tau}\ 0\ i\ fT = fT$
⟨*proof*⟩

lemma *liftp0* [*simp*]:
$\uparrow_p\ 0\ i\ p = p$
$\uparrow_{rp}\ 0\ i\ fs = fs$
$\uparrow_{fp}\ 0\ i\ f = f$
⟨*proof*⟩

lemma *lift0* [*simp*]:
$\uparrow\ 0\ i\ t = t$
$\uparrow_r\ 0\ i\ fs = fs$
$\uparrow_f\ 0\ i\ f = f$
⟨*proof*⟩

theorem *substT-liftT* [*simp*]:
$k \leq k' \Longrightarrow k' < k + n \Longrightarrow (\uparrow_\tau\ n\ k\ T)[k' \mapsto_\tau U]_\tau = \uparrow_\tau\ (n - 1)\ k\ T$
$k \leq k' \Longrightarrow k' < k + n \Longrightarrow (\uparrow_{r\tau}\ n\ k\ rT)[k' \mapsto_\tau U]_{r\tau} = \uparrow_{r\tau}\ (n - 1)\ k\ rT$
$k \leq k' \Longrightarrow k' < k + n \Longrightarrow (\uparrow_{f\tau}\ n\ k\ fT)[k' \mapsto_\tau U]_{f\tau} = \uparrow_{f\tau}\ (n - 1)\ k\ fT$
⟨*proof*⟩

theorem *liftT-substT* [*simp*]:
$k \leq k' \Longrightarrow \uparrow_\tau\ n\ k\ (T[k' \mapsto_\tau U]_\tau) = \uparrow_\tau\ n\ k\ T[k' + n \mapsto_\tau U]_\tau$
$k \leq k' \Longrightarrow \uparrow_{r\tau}\ n\ k\ (rT[k' \mapsto_\tau U]_{r\tau}) = \uparrow_{r\tau}\ n\ k\ rT[k' + n \mapsto_\tau U]_{r\tau}$
$k \leq k' \Longrightarrow \uparrow_{f\tau}\ n\ k\ (fT[k' \mapsto_\tau U]_{f\tau}) = \uparrow_{f\tau}\ n\ k\ fT[k' + n \mapsto_\tau U]_{f\tau}$
⟨*proof*⟩

theorem *liftT-substT'* [*simp*]:
$k' < k \Longrightarrow$
$\uparrow_\tau\ n\ k\ (T[k' \mapsto_\tau U]_\tau) = \uparrow_\tau\ n\ (k + 1)\ T[k' \mapsto_\tau \uparrow_\tau\ n\ (k - k')\ U]_\tau$
$k' < k \Longrightarrow$
$\uparrow_{r\tau}\ n\ k\ (rT[k' \mapsto_\tau U]_{r\tau}) = \uparrow_{r\tau}\ n\ (k + 1)\ rT[k' \mapsto_\tau \uparrow_\tau\ n\ (k - k')\ U]_{r\tau}$

$k' < k \Longrightarrow$
$\uparrow_{f\tau}\ n\ k\ (fT[k' \mapsto_\tau U]_{f\tau}) = \uparrow_{f\tau}\ n\ (k + 1)\ fT[k' \mapsto_\tau \uparrow_\tau\ n\ (k - k')\ U]_{f\tau}$
⟨*proof*⟩

lemma *liftT-substT-Top* [*simp*]:
$k \leq k' \Longrightarrow \uparrow_\tau\ n\ k'\ (T[k \mapsto_\tau Top]_\tau) = \uparrow_\tau\ n\ (Suc\ k')\ T[k \mapsto_\tau Top]_\tau$
$k \leq k' \Longrightarrow \uparrow_{r\tau}\ n\ k'\ (rT[k \mapsto_\tau Top]_{r\tau}) = \uparrow_{r\tau}\ n\ (Suc\ k')\ rT[k \mapsto_\tau Top]_{r\tau}$
$k \leq k' \Longrightarrow \uparrow_{f\tau}\ n\ k'\ (fT[k \mapsto_\tau Top]_{f\tau}) = \uparrow_{f\tau}\ n\ (Suc\ k')\ fT[k \mapsto_\tau Top]_{f\tau}$
⟨*proof*⟩

theorem *liftE-substE* [*simp*]:
$k \leq k' \Longrightarrow \uparrow_e\ n\ k\ (\Gamma[k' \mapsto_\tau U]_e) = \uparrow_e\ n\ k\ \Gamma[k' + n \mapsto_\tau U]_e$
⟨*proof*⟩

lemma *liftT-decT* [*simp*]:
$k \leq k' \Longrightarrow \uparrow_\tau\ n\ k'\ (\downarrow_\tau\ m\ k\ T) = \downarrow_\tau\ m\ k\ (\uparrow_\tau\ n\ (m + k')\ T)$
⟨*proof*⟩

lemma *liftT-substT-strange*:
$\uparrow_\tau\ n\ k\ T[n + k \mapsto_\tau U]_\tau = \uparrow_\tau\ n\ (Suc\ k)\ T[k \mapsto_\tau \uparrow_\tau\ n\ 0\ U]_\tau$
$\uparrow_{r\tau}\ n\ k\ rT[n + k \mapsto_\tau U]_{r\tau} = \uparrow_{r\tau}\ n\ (Suc\ k)\ rT[k \mapsto_\tau \uparrow_\tau\ n\ 0\ U]_{r\tau}$
$\uparrow_{f\tau}\ n\ k\ fT[n + k \mapsto_\tau U]_{f\tau} = \uparrow_{f\tau}\ n\ (Suc\ k)\ fT[k \mapsto_\tau \uparrow_\tau\ n\ 0\ U]_{f\tau}$
⟨*proof*⟩

lemma *liftp-liftp* [*simp*]:
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_p\ n'\ k'\ (\uparrow_p\ n\ k\ p) = \uparrow_p\ (n + n')\ k\ p$
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_{rp}\ n'\ k'\ (\uparrow_{rp}\ n\ k\ rp) = \uparrow_{rp}\ (n + n')\ k\ rp$
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_{fp}\ n'\ k'\ (\uparrow_{fp}\ n\ k\ fp) = \uparrow_{fp}\ (n + n')\ k\ fp$
⟨*proof*⟩

lemma *liftp-psize*[*simp*]:
$\|\uparrow_p\ n\ k\ p\|_p = \|p\|_p$
$\|\uparrow_{rp}\ n\ k\ fs\|_r = \|fs\|_r$
$\|\uparrow_{fp}\ n\ k\ f\|_f = \|f\|_f$
⟨*proof*⟩

lemma *lift-lift* [*simp*]:
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow\ n'\ k'\ (\uparrow\ n\ k\ t) = \uparrow\ (n + n')\ k\ t$
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_r\ n'\ k'\ (\uparrow_r\ n\ k\ fs) = \uparrow_r\ (n + n')\ k\ fs$
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_f\ n'\ k'\ (\uparrow_f\ n\ k\ f) = \uparrow_f\ (n + n')\ k\ f$
⟨*proof*⟩

lemma *liftE-liftE* [*simp*]:
$k \leq k' \Longrightarrow k' \leq k + n \Longrightarrow \uparrow_e\ n'\ k'\ (\uparrow_e\ n\ k\ \Gamma) = \uparrow_e\ (n + n')\ k\ \Gamma$
⟨*proof*⟩

lemma *liftE-liftE'* [*simp*]:
$i + m \leq j \Longrightarrow \uparrow_e\ n\ j\ (\uparrow_e\ m\ i\ \Gamma) = \uparrow_e\ m\ i\ (\uparrow_e\ n\ (j - m)\ \Gamma)$
⟨*proof*⟩

lemma *substT-substT*:
$i \leq j \Longrightarrow$
$T[Suc\ j \mapsto_\tau V]_\tau[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_\tau = T[i \mapsto_\tau U]_\tau[j \mapsto_\tau V]_\tau$
$i \leq j \Longrightarrow$
$rT[Suc\ j \mapsto_\tau V]_{r\tau}[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_{r\tau} = rT[i \mapsto_\tau U]_{r\tau}[j \mapsto_\tau V]_{r\tau}$
$i \leq j \Longrightarrow$
$fT[Suc\ j \mapsto_\tau V]_{f\tau}[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_{f\tau} = fT[i \mapsto_\tau U]_{f\tau}[j \mapsto_\tau V]_{f\tau}$
⟨*proof*⟩

lemma *substT-decT* [*simp*]:
$k \leq j \Longrightarrow (\downarrow_\tau i\ k\ T)[j \mapsto_\tau U]_\tau = \downarrow_\tau i\ k\ (T[i + j \mapsto_\tau U]_\tau)$
⟨*proof*⟩

lemma *substT-decT′* [*simp*]:
$i \leq j \Longrightarrow \downarrow_\tau k\ (Suc\ j)\ T[i \mapsto_\tau Top]_\tau = \downarrow_\tau k\ j\ (T[i \mapsto_\tau Top]_\tau)$
⟨*proof*⟩

lemma *substE-substE*:
$i \leq j \Longrightarrow \Gamma[Suc\ j \mapsto_\tau V]_e[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_e = \Gamma[i \mapsto_\tau U]_e[j \mapsto_\tau V]_e$
⟨*proof*⟩

lemma *substT-decE* [*simp*]:
$i \leq j \Longrightarrow \downarrow_e k\ (Suc\ j)\ \Gamma[i \mapsto_\tau Top]_e = \downarrow_e k\ j\ (\Gamma[i \mapsto_\tau Top]_e)$
⟨*proof*⟩

lemma *liftE-app* [*simp*]: $\uparrow_e n\ k\ (\Gamma\ @\ \Delta) = \uparrow_e n\ (k + \|\Delta\|)\ \Gamma\ @\ \uparrow_e n\ k\ \Delta$
⟨*proof*⟩

lemma *substE-app* [*simp*]:
$(\Gamma\ @\ \Delta)[k \mapsto_\tau T]_e = \Gamma[k + \|\Delta\| \mapsto_\tau T]_e\ @\ \Delta[k \mapsto_\tau T]_e$
⟨*proof*⟩

lemma *substs-app* [*simp*]: $t[k \mapsto_s ts\ @\ us] = t[k + \|us\| \mapsto_s ts][k \mapsto_s us]$
⟨*proof*⟩

theorem *decE-Nil* [*simp*]: $\downarrow_e n\ k\ [] = []$
⟨*proof*⟩

theorem *decE-Cons* [*simp*]:
$\downarrow_e n\ k\ (B :: \Gamma) = mapB\ (\downarrow_\tau n\ (k + \|\Gamma\|))\ B :: \downarrow_e n\ k\ \Gamma$
⟨*proof*⟩

theorem *decE-app* [*simp*]:
$\downarrow_e n\ k\ (\Gamma\ @\ \Delta) = \downarrow_e n\ (k + \|\Delta\|)\ \Gamma\ @\ \downarrow_e n\ k\ \Delta$
⟨*proof*⟩

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

$\langle proof \rangle$

theorem *decE-liftE* [*simp*]:
$k \leq k' \Longrightarrow k' + m \leq k + n \Longrightarrow \downarrow_e\ m\ k'\ (\uparrow_e\ n\ k\ \Gamma) = \uparrow_e\ (n - m)\ k\ \Gamma$
$\langle proof \rangle$

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

lemma *decT-decT* [*simp*]: $\downarrow_\tau\ n\ k\ (\downarrow_\tau\ n'\ (k + n)\ T) = \downarrow_\tau\ (n + n')\ k\ T$
$\langle proof \rangle$

lemma *decE-decE* [*simp*]: $\downarrow_e\ n\ k\ (\downarrow_e\ n'\ (k + n)\ \Gamma) = \downarrow_e\ (n + n')\ k\ \Gamma$
$\langle proof \rangle$

lemma *decE-length* [*simp*]: $\|\downarrow_e\ n\ k\ \Gamma\| = \|\Gamma\|$
$\langle proof \rangle$

lemma *liftrT-assoc-None* [*simp*]: $(\uparrow_{r\tau}\ n\ k\ fs\langle l\rangle_? = \bot) = (fs\langle l\rangle_? = \bot)$
$\langle proof \rangle$

lemma *liftrT-assoc-Some*: $fs\langle l\rangle_? = \lfloor T\rfloor \Longrightarrow \uparrow_{r\tau}\ n\ k\ fs\langle l\rangle_? = \lfloor\uparrow_\tau\ n\ k\ T\rfloor$
$\langle proof \rangle$

lemma *liftrp-assoc-None* [*simp*]: $(\uparrow_{rp}\ n\ k\ fps\langle l\rangle_? = \bot) = (fps\langle l\rangle_? = \bot)$
$\langle proof \rangle$

lemma *liftr-assoc-None* [*simp*]: $(\uparrow_r\ n\ k\ fs\langle l\rangle_? = \bot) = (fs\langle l\rangle_? = \bot)$
$\langle proof \rangle$

lemma *unique-liftrT* [*simp*]: $unique\ (\uparrow_{r\tau}\ n\ k\ fs) = unique\ fs$
$\langle proof \rangle$

lemma *substrTT-assoc-None* [*simp*]: $(fs[k \mapsto_\tau\ U]_{r\tau}\langle a\rangle_? = \bot) = (fs\langle a\rangle_? = \bot)$
$\langle proof \rangle$

lemma *substrTT-assoc-Some* [*simp*]:
$fs\langle a\rangle_? = \lfloor T\rfloor \Longrightarrow fs[k \mapsto_\tau\ U]_{r\tau}\langle a\rangle_? = \lfloor T[k \mapsto_\tau\ U]_\tau\rfloor$
$\langle proof \rangle$

lemma *substrT-assoc-None* [*simp*]: $(fs[k \mapsto_\tau\ P]_r\langle l\rangle_? = \bot) = (fs\langle l\rangle_? = \bot)$
$\langle proof \rangle$

lemma *substrp-assoc-None* [*simp*]: $(fps[k \mapsto_\tau\ U]_{rp}\langle l\rangle_? = \bot) = (fps\langle l\rangle_? = \bot)$
$\langle proof \rangle$

lemma *substr-assoc-None* [*simp*]: $(fs[k \mapsto\ u]_r\langle l\rangle_? = \bot) = (fs\langle l\rangle_? = \bot)$
$\langle proof \rangle$

lemma *unique-substrT* [*simp*]: $unique\ (fs[k \mapsto_\tau\ U]_{r\tau}) = unique\ fs$
⟨*proof*⟩

lemma *liftrT-set*: $(a,\ T) \in set\ fs \Longrightarrow (a,\ \uparrow_\tau\ n\ k\ T) \in set\ (\uparrow_{r\tau}\ n\ k\ fs)$
⟨*proof*⟩

lemma *liftrT-setD*:
$(a,\ T) \in set\ (\uparrow_{r\tau}\ n\ k\ fs) \Longrightarrow \exists\, T'.\ (a,\ T') \in set\ fs \wedge T = \uparrow_\tau\ n\ k\ T'$
⟨*proof*⟩

lemma *substrT-set*: $(a,\ T) \in set\ fs \Longrightarrow (a,\ T[k \mapsto_\tau\ U]_\tau) \in set\ (fs[k \mapsto_\tau\ U]_{r\tau})$
⟨*proof*⟩

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

3.3 Well-formedness

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

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

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

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

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

lemma *wf-TVarB*: $\Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow \mathit{TVarB}\ T :: \Gamma \vdash_{wf}$
⟨*proof*⟩

lemma *wf-VarB*: $\Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} \Longrightarrow \mathit{VarB}\ T :: \Gamma \vdash_{wf}$
⟨*proof*⟩

lemma *map-is-TVarb*:
$\mathit{map}\ \mathit{is\text{-}TVarB}\ \Gamma' = \mathit{map}\ \mathit{is\text{-}TVarB}\ \Gamma \Longrightarrow$
$\Gamma\langle i\rangle = \lfloor \mathit{TVarB}\ T\rfloor \Longrightarrow \exists\, T.\ \Gamma'\langle i\rangle = \lfloor \mathit{TVarB}\ T\rfloor$
⟨*proof*⟩

lemma *wf-equallength*:
assumes H: $\Gamma \vdash_{wf} T$
shows $\mathit{map}\ \mathit{is\text{-}TVarB}\ \Gamma' = \mathit{map}\ \mathit{is\text{-}TVarB}\ \Gamma \Longrightarrow \Gamma' \vdash_{wf} T$ ⟨*proof*⟩

lemma *wfE-replace*:
$\Delta\ @\ B :: \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf\,B} B' \Longrightarrow \mathit{is\text{-}TVarB}\ B' = \mathit{is\text{-}TVarB}\ B \Longrightarrow$
$\Delta\ @\ B' :: \Gamma \vdash_{wf}$
⟨*proof*⟩

lemma *wf-weaken*:
assumes H: $\Delta\ @\ \Gamma \vdash_{wf} T$
shows $\uparrow_e (\mathit{Suc}\ 0)\ 0\ \Delta\ @\ B :: \Gamma \vdash_{wf} \uparrow_\tau (\mathit{Suc}\ 0)\ \|\Delta\|\ T$
⟨*proof*⟩

lemma *wf-weaken'*: $\Gamma \vdash_{wf} T \Longrightarrow \Delta\ @\ \Gamma \vdash_{wf} \uparrow_\tau \|\Delta\|\ 0\ T$
⟨*proof*⟩

lemma *wfE-weaken*: $\Delta\ @\ \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf\,B} B \Longrightarrow \uparrow_e (\mathit{Suc}\ 0)\ 0\ \Delta\ @\ B :: \Gamma \vdash_{wf}$
⟨*proof*⟩

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

theorem *wf-subst*:
$\Delta\ @\ B :: \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Delta[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf} T[\|\Delta\| \mapsto_\tau U]_\tau$
$\forall (l,\ T) \in \mathit{set}\ (\mathit{rT}{::}\mathit{rcdT}).\ \Delta\ @\ B :: \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow$
$\forall (l,\ T) \in \mathit{set}\ \mathit{rT}.\ \Delta[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf} T[\|\Delta\| \mapsto_\tau U]_\tau$
$\Delta\ @\ B :: \Gamma \vdash_{wf} \mathit{snd}\ (\mathit{fT}{::}\mathit{fldT}) \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow$
$\Delta[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf} \mathit{snd}\ \mathit{fT}[\|\Delta\| \mapsto_\tau U]_\tau$
⟨*proof*⟩

theorem *wf-dec*: $\Delta\ @\ \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} \downarrow_\tau \|\Delta\|\ 0\ T$
⟨*proof*⟩

theorem *wfE-subst*: $\Delta\ @\ B :: \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Delta[0 \mapsto_\tau U]_e\ @\ \Gamma \vdash_{wf}$
⟨*proof*⟩

3.4 Subtyping

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

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

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

lemma *wf-subtype*:
  assumes PQ: $\Gamma \vdash P <: Q$
  shows $\Gamma \vdash_{wf} P \land \Gamma \vdash_{wf} Q$ ⟨*proof*⟩

lemma *wf-subtypeE*:
  assumes H: $\Gamma \vdash T <: U$
  and H': $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow P$
  shows P
  ⟨*proof*⟩

lemma *subtype-refl*: — A.1
  $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash T <: T$
  $\Gamma \vdash_{wf} \Longrightarrow \forall (l::name,\ T) \in set\ fTs.\ \Gamma \vdash_{wf} T \longrightarrow \Gamma \vdash T <: T$
  $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} snd\ (fT::fldT) \Longrightarrow \Gamma \vdash snd\ fT <: snd\ fT$
  ⟨*proof*⟩

lemma *subtype-weaken*:

assumes H: Δ @ $\Gamma \vdash P <: Q$
and wf: $\Gamma \vdash_{wfB} B$
shows $\uparrow_e$ 1 0 Δ @ B :: $\Gamma \vdash \uparrow_\tau 1 \; \|\Delta\| \; P <: \uparrow_\tau 1 \; \|\Delta\| \; Q$ ⟨*proof*⟩

lemma *subtype-weaken'*: — A.2
$\Gamma \vdash P <: Q \Longrightarrow \Delta$ @ $\Gamma \vdash_{wf} \Longrightarrow \Delta$ @ $\Gamma \vdash \uparrow_\tau \|\Delta\| \; 0 \; P <: \uparrow_\tau \|\Delta\| \; 0 \; Q$
⟨*proof*⟩

lemma *fieldT-size* [*simp*]:
$(a, T) \in set\ fs \Longrightarrow size\ T < Suc\ (size\text{-}list\ (size\text{-}prod\ (\lambda x.\ 0)\ size)\ fs)$
⟨*proof*⟩

lemma *subtype-trans*: — A.3
$\Gamma \vdash S <: Q \Longrightarrow \Gamma \vdash Q <: T \Longrightarrow \Gamma \vdash S <: T$
Δ @ $TVarB\ Q$ :: $\Gamma \vdash M <: N \Longrightarrow \Gamma \vdash P <: Q \Longrightarrow$
Δ @ $TVarB\ P$ :: $\Gamma \vdash M <: N$
⟨*proof*⟩

lemma *substT-subtype*: — A.10
assumes H: Δ @ $TVarB\ Q$ :: $\Gamma \vdash S <: T$
shows $\Gamma \vdash P <: Q \Longrightarrow \Delta[0 \mapsto_\tau P]_e$ @ $\Gamma \vdash S[\|\Delta\| \mapsto_\tau P]_\tau <: T[\|\Delta\| \mapsto_\tau P]_\tau$
⟨*proof*⟩

lemma *subst-subtype*:
assumes H: Δ @ $VarB\ V$ :: $\Gamma \vdash T <: U$
shows $\downarrow_e$ 1 0 Δ @ $\Gamma \vdash \downarrow_\tau 1 \; \|\Delta\| \; T <: \downarrow_\tau 1 \; \|\Delta\| \; U$
⟨*proof*⟩

3.5 Typing

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

inductive
ptyping :: *pat* ⇒ *type* ⇒ *env* ⇒ *bool* (‹⊢ - : - ⇒ -› [*50*, *50*, *50*] *50*)
and *ptypings* :: *rpat* ⇒ *rcdT* ⇒ *env* ⇒ *bool* (‹⊢ - [:] - ⇒ -› [*50*, *50*, *50*] *50*)
where
P-Var: $\vdash PVar\ T : T \Rightarrow [VarB\ T]$
| *P-Rcd*: $\vdash fps$ [:] $fTs \Rightarrow \Delta \Longrightarrow \vdash PRcd\ fps : RcdT\ fTs \Rightarrow \Delta$
| *P-Nil*: $\vdash$ [] [:] [] $\Rightarrow$ []
| *P-Cons*: $\vdash p : T \Rightarrow \Delta_1 \Longrightarrow \vdash fps$ [:] $fTs \Rightarrow \Delta_2 \Longrightarrow fps\langle l\rangle_? = \bot \Longrightarrow$
$\vdash ((l, p) :: fps)$ [:] $((l, T) :: fTs) \Rightarrow \uparrow_e \|\Delta_1\| \; 0 \; \Delta_2$ @ Δ_1

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

inductive
  typing :: *env* ⇒ *trm* ⇒ *type* ⇒ *bool* (‹- ⊢ - : -› [*50*, *50*, *50*] *50*)
  and *typings* :: *env* ⇒ *rcd* ⇒ *rcdT* ⇒ *bool* (‹- ⊢ - [:] -› [*50*, *50*, *50*] *50*)
where
  $T\text{-}Var$: $\Gamma \vdash_{wf} \Longrightarrow \Gamma\langle i\rangle = \lfloor VarB\ U\rfloor \Longrightarrow T = \uparrow_\tau (Suc\ i)\ 0\ U \Longrightarrow \Gamma \vdash Var\ i : T$
| $T\text{-}Abs$: $VarB\ T_1 :: \Gamma \vdash t_2 : T_2 \Longrightarrow \Gamma \vdash (\lambda{:}T_1.\ t_2) : T_1 \to \downarrow_\tau 1\ 0\ T_2$
| $T\text{-}App$: $\Gamma \vdash t_1 : T_{11} \to T_{12} \Longrightarrow \Gamma \vdash t_2 : T_{11} \Longrightarrow \Gamma \vdash t_1 \cdot t_2 : T_{12}$
| $T\text{-}TAbs$: $TVarB\ T_1 :: \Gamma \vdash t_2 : T_2 \Longrightarrow \Gamma \vdash (\lambda{<:}T_1.\ t_2) : (\forall{<:}T_1.\ T_2)$
| $T\text{-}TApp$: $\Gamma \vdash t_1 : (\forall{<:}T_{11}.\ T_{12}) \Longrightarrow \Gamma \vdash T_2 <: T_{11} \Longrightarrow$
    $\Gamma \vdash t_1 \cdot_\tau T_2 : T_{12}[0 \mapsto_\tau T_2]_\tau$
| $T\text{-}Sub$: $\Gamma \vdash t : S \Longrightarrow \Gamma \vdash S <: T \Longrightarrow \Gamma \vdash t : T$
| $T\text{-}Let$: $\Gamma \vdash t_1 : T_1 \Longrightarrow \vdash p : T_1 \Rightarrow \Delta \Longrightarrow \Delta\ @\ \Gamma \vdash t_2 : T_2 \Longrightarrow$
    $\Gamma \vdash (LET\ p = t_1\ IN\ t_2) : \downarrow_\tau \|\Delta\|\ 0\ T_2$
| $T\text{-}Rcd$: $\Gamma \vdash fs\ [:]\ fTs \Longrightarrow \Gamma \vdash Rcd\ fs : RcdT\ fTs$
| $T\text{-}Proj$: $\Gamma \vdash t : RcdT\ fTs \Longrightarrow fTs\langle l\rangle_? = \lfloor T\rfloor \Longrightarrow \Gamma \vdash t..l : T$
| $T\text{-}Nil$: $\Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash []\ [:]\ []$
| $T\text{-}Cons$: $\Gamma \vdash t : T \Longrightarrow \Gamma \vdash fs\ [:]\ fTs \Longrightarrow fs\langle l\rangle_? = \bot \Longrightarrow$
    $\Gamma \vdash (l, t) :: fs\ [:]\ (l, T) :: fTs$

theorem *wf-typeE1*:
  $\Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wf}$
  $\Gamma \vdash fs\ [:]\ fTs \Longrightarrow \Gamma \vdash_{wf}$
  ⟨*proof*⟩

theorem *wf-typeE2*:
  $\Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wf} T$
  $\Gamma' \vdash fs\ [:]\ fTs \Longrightarrow (\forall (l, T) \in set\ fTs.\ \Gamma' \vdash_{wf} T) \land$
    $unique\ fTs \land (\forall l.\ (fs\langle l\rangle_? = \bot) = (fTs\langle l\rangle_? = \bot))$
⟨*proof*⟩

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

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

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

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

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

lemma *narrow-type*: — A.7
Δ @ *TVarB* Q :: $\Gamma \vdash t : T \Longrightarrow$
$\Gamma \vdash P <: Q \Longrightarrow \Delta$ @ *TVarB* P :: $\Gamma \vdash t : T$
Δ @ *TVarB* Q :: $\Gamma \vdash$ *ts* [:] *Ts* $\Longrightarrow$
$\Gamma \vdash P <: Q \Longrightarrow \Delta$ @ *TVarB* P :: $\Gamma \vdash$ *ts* [:] *Ts*
⟨*proof*⟩

lemma *typings-setD*:
assumes H: $\Gamma \vdash$ *fs* [:] *fTs*
shows $(l,\ T) \in$ *set fTs* $\Longrightarrow \exists\, t.\ fs\langle l\rangle_? = \lfloor t \rfloor \wedge \Gamma \vdash t : T$
⟨*proof*⟩

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

lemma *Abs-type*: — A.13(1)
assumes H: $\Gamma \vdash (\lambda{:}S.\ s) : T$
shows $\Gamma \vdash T <: U \rightarrow U' \Longrightarrow$
$(\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow$ *VarB* S :: $\Gamma \vdash s : S' \Longrightarrow$
$\Gamma \vdash \downarrow_\tau\ 1\ 0\ S' <: U' \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *Abs-type'*:
assumes $\Gamma \vdash (\lambda{:}S.\ s) : U \rightarrow U'$
and $\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow$ *VarB* S :: $\Gamma \vdash s : S' \Longrightarrow \Gamma \vdash \downarrow_\tau\ 1\ 0\ S' <: U' \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *TAbs-type*: — A.13(2)
assumes H: $\Gamma \vdash (\lambda{<:}S.\ s) : T$
shows $\Gamma \vdash T <: (\forall{<:}U.\ U') \Longrightarrow$
$(\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow$ *TVarB* U :: $\Gamma \vdash s : S' \Longrightarrow$
TVarB U :: $\Gamma \vdash S' <: U' \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *TAbs-type'*:
assumes $\Gamma \vdash (\lambda{<:}S.\ s) : (\forall{<:}U.\ U')$
and $\bigwedge S'.\ \Gamma \vdash U <: S \Longrightarrow$ *TVarB* U :: $\Gamma \vdash s : S' \Longrightarrow$ *TVarB* U :: $\Gamma \vdash S' <: U'$
$\Longrightarrow P$
shows P
⟨*proof*⟩

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

lemma *Rcd-type1*: — A.13(3)
assumes $\Gamma \vdash t : T$
shows $t = Rcd\ fs \Longrightarrow \Gamma \vdash T <: RcdT\ fTs \Longrightarrow$
$\forall (l,\ U) \in set\ fTs.\ \exists\, u.\ fs\langle l\rangle_? = \lfloor u\rfloor \wedge \Gamma \vdash u : U$
⟨*proof*⟩

lemma *Rcd-type1'*:
assumes H: $\Gamma \vdash Rcd\ fs : RcdT\ fTs$
shows $\forall (l,\ U) \in set\ fTs.\ \exists\, u.\ fs\langle l\rangle_? = \lfloor u\rfloor \wedge \Gamma \vdash u : U$
⟨*proof*⟩

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

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

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

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

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

lemma *Rcd-type2-aux*:
$[\![\Gamma \vdash T <: RcdT\ fTs;\ \forall (l,\ U){\in}set\ fTs.\ \exists\, u.\ fs\langle l\rangle_? = \lfloor u\rfloor \wedge \Gamma \vdash u : U]\!]$
$\Longrightarrow \Gamma \vdash map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs\ [:]\ fTs$
⟨*proof*⟩

lemma *Rcd-type2*:
$\Gamma \vdash Rcd\ fs : T \Longrightarrow \Gamma \vdash T <: RcdT\ fTs \Longrightarrow$
$\Gamma \vdash map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs\ [:]\ fTs$
⟨*proof*⟩

lemma *Rcd-type2'*:
assumes H: $\Gamma \vdash Rcd\ fs : RcdT\ fTs$
shows $\Gamma \vdash map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs\ [:]\ fTs$
⟨*proof*⟩

lemma *T-eq*: $\Gamma \vdash t : T \Longrightarrow T = T' \Longrightarrow \Gamma \vdash t : T'$ ⟨*proof*⟩

lemma *ptyping-length* [*simp*]:
$\vdash p : T \Rightarrow \Delta \Longrightarrow \|p\|_p = \|\Delta\|$
$\vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow \|fps\|_r = \|\Delta\|$
⟨*proof*⟩

lemma *lift-ptyping*:

$\vdash p : T \Rightarrow \Delta \Longrightarrow \vdash \uparrow_p\ n\ k\ p : \uparrow_\tau\ n\ k\ T \Rightarrow \uparrow_e\ n\ k\ \Delta$
$\vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow \vdash \uparrow_{rp}\ n\ k\ fps\ [:]\ \uparrow_{r\tau}\ n\ k\ fTs \Rightarrow \uparrow_e\ n\ k\ \Delta$
$\langle proof \rangle$

lemma *type-weaken*:
$\Delta\ @\ \Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wf B}\ B \Longrightarrow$
$\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma \vdash \uparrow\ 1\ \|\Delta\|\ t : \uparrow_\tau\ 1\ \|\Delta\|\ T$
$\Delta\ @\ \Gamma \vdash fs\ [:]\ fTs \Longrightarrow \Gamma \vdash_{wf B}\ B \Longrightarrow$
$\uparrow_e\ 1\ 0\ \Delta\ @\ B :: \Gamma \vdash \uparrow_r\ 1\ \|\Delta\|\ fs\ [:]\ \uparrow_{r\tau}\ 1\ \|\Delta\|\ fTs$
$\langle proof \rangle$

lemma *type-weaken'*: — A.5(6)
$\Gamma \vdash t : T \Longrightarrow \Delta\ @\ \Gamma \vdash_{wf} \Longrightarrow \Delta\ @\ \Gamma \vdash \uparrow\ \|\Delta\|\ 0\ t : \uparrow_\tau\ \|\Delta\|\ 0\ T$
$\langle proof \rangle$

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

lemma *subst-ptyping*:
$\vdash p : T \Rightarrow \Delta \Longrightarrow \vdash p[k \mapsto_\tau U]_p : T[k \mapsto_\tau U]_\tau \Rightarrow \Delta[k \mapsto_\tau U]_e$
$\vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow \vdash fps[k \mapsto_\tau U]_{rp}\ [:]\ fTs[k \mapsto_\tau U]_{r\tau} \Rightarrow \Delta[k \mapsto_\tau U]_e$
$\langle proof \rangle$

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

theorem *substT-type*: — A.11
$\Delta\ @\ TVarB\ Q :: \Gamma \vdash t : T \Longrightarrow \Gamma \vdash P <: Q \Longrightarrow$
$\Delta[0 \mapsto_\tau P]_e\ @\ \Gamma \vdash t[\|\Delta\| \mapsto_\tau P] : T[\|\Delta\| \mapsto_\tau P]_\tau$
$\Delta\ @\ TVarB\ Q :: \Gamma \vdash fs\ [:]\ fTs \Longrightarrow \Gamma \vdash P <: Q \Longrightarrow$
$\Delta[0 \mapsto_\tau P]_e\ @\ \Gamma \vdash fs[\|\Delta\| \mapsto_\tau P]_r\ [:]\ fTs[\|\Delta\| \mapsto_\tau P]_{r\tau}$
$\langle proof \rangle$

3.6 Evaluation

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

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

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

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

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

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

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

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

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

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

$\langle proof \rangle$

lemma *matchs-eq*:
assumes H: $\vdash fps\ [\rhd]\ fs \Rightarrow ts$
shows $\forall (l, p) \in set\ fps.\ fs\langle l\rangle_? = fs'\langle l\rangle_? \Longrightarrow \vdash fps\ [\rhd]\ fs' \Rightarrow ts$
$\langle proof \rangle$

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

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

lemma *matchs-reorder'*:
$\vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow \forall (l, U) \in set\ fTs.\ \exists u.\ fs\langle l\rangle_? = \lfloor u \rfloor \Longrightarrow$
$\vdash fps\ [\rhd]\ map\ (\lambda(l, T).\ (l, the\ (fs\langle l\rangle_?)))\ fTs \Rightarrow ts \Longrightarrow \vdash fps\ [\rhd]\ fs \Rightarrow ts$
$\langle proof \rangle$

theorem *matchs-tl*:
assumes H: $\vdash fps\ [\rhd]\ (l, t) :: fs \Rightarrow ts$
shows $fps\langle l\rangle_? = \bot \Longrightarrow \vdash fps\ [\rhd]\ fs \Rightarrow ts$
$\langle proof \rangle$

theorem *match-length*:
$\vdash p \rhd t \Rightarrow ts \Longrightarrow \vdash p : T \Rightarrow \Delta \Longrightarrow \|ts\| = \|\Delta\|$
$\vdash fps\ [\rhd]\ ft \Rightarrow ts \Longrightarrow \vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow \|ts\| = \|\Delta\|$
$\langle proof \rangle$

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

theorem *match-type*: — A.17
$\vdash p : T_1 \Rightarrow \Delta \Longrightarrow \Gamma_2 \vdash t_1 : T_1 \Longrightarrow$
$\Gamma_1 @ \Delta @ \Gamma_2 \vdash t_2 : T_2 \Longrightarrow \vdash p \rhd t_1 \Rightarrow ts \Longrightarrow$
$\downarrow_e \|\Delta\|\ 0\ \Gamma_1 @ \Gamma_2 \vdash t_2[\|\Gamma_1\| \mapsto_s ts] : \downarrow_\tau \|\Delta\|\ \|\Gamma_1\|\ T_2$
$\vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow \Gamma_2 \vdash fs\ [:]\ fTs \Longrightarrow$
$\Gamma_1 @ \Delta @ \Gamma_2 \vdash t_2 : T_2 \Longrightarrow \vdash fps\ [\rhd]\ fs \Rightarrow ts \Longrightarrow$
$\downarrow_e \|\Delta\|\ 0\ \Gamma_1 @ \Gamma_2 \vdash t_2[\|\Gamma_1\| \mapsto_s ts] : \downarrow_\tau \|\Delta\|\ \|\Gamma_1\|\ T_2$
$\langle proof \rangle$

lemma *evals-labels* [*simp*]:
assumes H: $fs\ [\longmapsto]\ fs'$

shows $(fs\langle l\rangle_? = \bot) = (fs'\langle l\rangle_? = \bot)$ ⟨*proof*⟩

theorem *preservation*: — A.20
$\Gamma \vdash t : T \Longrightarrow t \longmapsto t' \Longrightarrow \Gamma \vdash t' : T$
$\Gamma \vdash fs\ [:]\ fTs \Longrightarrow fs\ [\longmapsto]\ fs' \Longrightarrow \Gamma \vdash fs'\ [:]\ fTs$
⟨*proof*⟩

lemma *Fun-canonical*: — A.14(1)
assumes *ty*: $[] \vdash v : T_1 \to T_2$
shows $v \in value \Longrightarrow \exists\, t\ S.\ v = (\lambda{:}S.\ t)$ ⟨*proof*⟩

lemma *TyAll-canonical*: — A.14(3)
assumes *ty*: $[] \vdash v : (\forall{<:}T_1.\ T_2)$
shows $v \in value \Longrightarrow \exists\, t\ S.\ v = (\lambda{<:}S.\ t)$ ⟨*proof*⟩

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

lemma *RcdT-canonical*: — A.14(2)
assumes *ty*: $[] \vdash v : RcdT\ fTs$
shows $v \in value \Longrightarrow$
$\exists fs.\ v = Rcd\ fs \wedge (\forall (l,\ t) \in set\ fs.\ t \in value)$ ⟨*proof*⟩

theorem *reorder-prop*:
$\forall (l,\ t) \in set\ fs.\ P\ t \Longrightarrow \forall (l,\ U) {\in} set\ fTs.\ \exists\, u.\ fs\langle l\rangle_? = \lfloor u \rfloor \Longrightarrow$
$\forall (l,\ t) \in set\ (map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs).\ P\ t$
⟨*proof*⟩

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

theorem *ptyping-match*:
$\vdash p : T \Rightarrow \Delta \Longrightarrow [] \vdash t : T \Longrightarrow t \in value \Longrightarrow$
$\exists\, ts.\ \vdash p \rhd t \Rightarrow ts$
$\vdash fps\ [:]\ fTs \Rightarrow \Delta \Longrightarrow [] \vdash fs\ [:]\ fTs \Longrightarrow$
$\forall (l,\ t) \in set\ fs.\ t \in value \Longrightarrow \exists\, us.\ \vdash fps\ [\rhd]\ fs \Rightarrow us$
⟨*proof*⟩

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*⟩

4 Evaluation contexts

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

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

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

lemma *rctxt-labels*:
  assumes *H*: $E \in rctxt$
  shows $E\ t\langle l\rangle_? = \bot \Longrightarrow E\ t'\langle l\rangle_? = \bot$ ⟨*proof*⟩

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

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

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

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

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 $\Gamma \vdash t_{12}[0 \mapsto t_2] : T$
⟨*proof*⟩

theorem *TAbs-preservation*: — A.19(2)
assumes H: $\Gamma \vdash (\lambda{<:}T_{11}.\ t_{12}) \cdot_\tau T_2 : T$
shows $\Gamma \vdash t_{12}[0 \mapsto_\tau T_2] : T$
⟨*proof*⟩

theorem *Let-preservation*: — A.19(3)
assumes H: $\Gamma \vdash (LET\ p = t_1\ IN\ t_2) : T$
shows $\vdash p \rhd t_1 \Rightarrow ts \Longrightarrow \Gamma \vdash t_2[0 \mapsto_s ts] : T$
⟨*proof*⟩

theorem *Proj-preservation*: — A.19(4)
assumes H: $\Gamma \vdash Rcd\ fs..l : T$
shows $fs\langle l\rangle_? = \lfloor v \rfloor \Longrightarrow \Gamma \vdash v : T$
⟨*proof*⟩

theorem *preservation*: — A.20
assumes H: $t \longmapsto_c t'$
shows $\Gamma \vdash t : T \Longrightarrow \Gamma \vdash t' : T$ ⟨*proof*⟩

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

theorem *context-decomp*: — A.15
$[] \vdash t : T \Longrightarrow$
$t \in value \lor (\exists E\ t_0\ t_0'.\ E \in ctxt \land t = E\ t_0 \land t_0 \longmapsto_c t_0')$
$[] \vdash fs\ [:]\ fTs \Longrightarrow$
$(\forall (l, t) \in set\ fs.\ t \in value) \lor (\exists E\ t_0\ t_0'.\ E \in rctxt \land fs = E\ t_0 \land t_0 \longmapsto_c t_0')$
⟨*proof*⟩

theorem *progress*: — A.16
  assumes H: $[] \vdash t : T$
  shows $t \in value \vee (\exists t'.\ t \longmapsto_c t')$
⟨*proof*⟩

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

lemma *ctxt-imp-eval*:
  $E \in ctxt \Longrightarrow t \longmapsto t' \Longrightarrow E\ t \longmapsto E\ t'$
  $E_r \in rctxt \Longrightarrow t \longmapsto t' \Longrightarrow E_r\ t\ [\longmapsto]\ E_r\ t'$
  ⟨*proof*⟩

lemma *eval-evalc-eq*: $(t \longmapsto t') = (t \longmapsto_c t')$
⟨*proof*⟩

5 Executing the specification

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

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

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

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

lemma [*code-pred-intro Rcd-Nil*]: *valuep* (*Rcd* [])
⟨*proof*⟩

lemma [*code-pred-intro Rcd-Cons*]: *valuep t* ⟹ *valuep* (*Rcd fs*) ⟹ *valuep* (*Rcd* ((*l*, *t*) # *fs*))
⟨*proof*⟩

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

code-pred (*modes*: *i* => *bool*) *valuep*

⟨*proof*⟩

thm *valuep.equation*

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

thm *norm.equation*

lemma [*code*]:
normal-forms = *set-of-pred o normalize*
⟨*proof*⟩

lemma [*code-unfold*]: $x \in$ *value* $\longleftrightarrow$ *valuep x*
⟨*proof*⟩

definition
natT :: *type* **where**
natT ≡ ∀<:*Top*. (∀<:*TVar 0*. (∀<:*TVar 1*. (*TVar 2* → *TVar 1*) → *TVar 0* → *TVar 1*))

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

value *normal-forms fact2*

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

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

References

[1] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The POPLMARK Challenge. In T. Melham and J. Hurd, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2005*, LNCS. Springer-Verlag, 2005.

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

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