

A Solution to the POPLMARK Challenge in Isabelle/HOL

Stefan Berghofer

December 12, 2009

Abstract

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

Contents

1	General Utilities	2
2	Formalization of the basic calculus	3
2.1	Types and Terms	4
2.2	Lifting and Substitution	5
2.3	Well-formedness	8
2.4	Subtyping	10
2.5	Typing	12
2.6	Evaluation	14
3	Extending the calculus with records	16
3.1	Types and Terms	16
3.2	Lifting and Substitution	17
3.3	Well-formedness	25
3.4	Subtyping	27
3.5	Typing	29
3.6	Evaluation	33
4	Evaluation contexts	36
5	Executing the specification	38

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*:

$$\begin{aligned} (\text{True} \implies \text{PROP } P) &\equiv \text{PROP } P \\ (\text{PROP } P \implies \text{True}) &\equiv \text{PROP } \text{Trueprop } \text{True} \\ (\bigwedge x. \text{True}) &\equiv \text{PROP } \text{Trueprop } \text{True} \\ \langle \text{proof} \rangle \end{aligned}$$

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

lemma *ballpI*: $(\bigwedge x y. (x, y) \in A \implies P x y) \implies \forall (x, y) \in A. P x y$
 $\langle \text{proof} \rangle$

lemma *bpspec*: $\forall (x, y) \in A. P x y \implies (x, y) \in A \implies P x y$
 $\langle \text{proof} \rangle$

lemma *ballpE*: $\forall (x, y) \in A. P x y \implies (P x y \implies Q) \implies$
 $((x, y) \notin A \implies Q) \implies Q$
 $\langle \text{proof} \rangle$

lemma *bexpI*: $P x y \implies (x, y) \in A \implies \exists (x, y) \in A. P x y$
 $\langle \text{proof} \rangle$

lemma *bexpE*: $\exists (x, y) \in A. P x y \implies$
 $(\bigwedge x y. (x, y) \in A \implies P x y \implies Q) \implies Q$
 $\langle \text{proof} \rangle$

lemma *ball-eq-sym*: $\forall (x, y) \in S. f x y = g x y \implies \forall (x, y) \in S. g x y = f x y$
 $\langle \text{proof} \rangle$

lemma *measure-eq [simp]*: $(x, y) \in \text{measure } f = (f x < f y)$
 $\langle \text{proof} \rangle$

lemma *wf-measure-size*: *wf (measure size)* $\langle \text{proof} \rangle$

notation

Some ($\lfloor - \rfloor$)

notation

None (\perp)

notation

length ($\| - \|$)

notation

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

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

primrec

nth-el :: 'a list \Rightarrow nat \Rightarrow 'a option (-(-) [90, 0] 91)

where

$\llbracket i \rrbracket = \perp$

| $(x \# xs)\langle i \rangle = (\text{case } i \text{ of } 0 \Rightarrow [x] \mid \text{Suc } j \Rightarrow xs \langle j \rangle)$

lemma [simp]: $i < \|xs\| \Longrightarrow (xs @ ys)\langle i \rangle = xs\langle i \rangle$

<proof>

lemma [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

$\llbracket a \rrbracket? = \perp$

| $(x \# xs)\langle a \rangle? = (\text{if } \text{fst } x = a \text{ then } [\text{snd } x] \text{ else } xs\langle a \rangle?)$

primrec *unique* :: ('a \times 'b) list \Rightarrow bool

where

unique $\llbracket \rrbracket = \text{True}$

| *unique* $(x \# xs) = (xs\langle \text{fst } x \rangle? = \perp \wedge \text{unique } xs)$

lemma *assoc-set*: $ps\langle x \rangle? = [y] \Longrightarrow (x, y) \in \text{set } ps$

<proof>

lemma *map-assoc-None* [simp]:

$ps\langle x \rangle? = \perp \Longrightarrow \text{map } (\lambda(x, y). (x, f x y)) ps\langle x \rangle? = \perp$

<proof>

no-syntax

-ofsort :: [tid, sort] \Rightarrow type (-:- [1000, 0] 1000)

-constrain :: [logic, type] \Rightarrow 'a (-:- [4, 0] 3)

-idtyp :: [id, type] \Rightarrow idt (-:- $\llbracket \rrbracket 0$)

-idtypdummy :: type \Rightarrow idt ('(-):- $\llbracket \rrbracket 0$)

-Map :: maplets \Rightarrow 'a $\sim \Rightarrow$ 'b ((1[-]))

2 Formalization of the basic calculus

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

2.1 Types and Terms

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

```
datatype type =
  TVar nat
  | Top
  | Fun type type  (infixr  $\rightarrow$  200)
  | TyAll type type (( $\exists\forall<:-./$  -) [0, 10] 10)
```

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

```
datatype binding = VarB type | TVarB type
types env = binding list
```

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

```
primrec is-TVarB :: binding  $\Rightarrow$  bool
where
  is-TVarB (VarB T) = False
  | is-TVarB (TVarB T) = True
```

```
primrec type-ofB :: binding  $\Rightarrow$  type
where
  type-ofB (VarB T) = T
  | type-ofB (TVarB T) = T
```

```
primrec mapB :: (type  $\Rightarrow$  type)  $\Rightarrow$  binding  $\Rightarrow$  binding
where
  mapB f (VarB T) = VarB (f T)
  | mapB f (TVarB T) = TVarB (f T)
```

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

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

2.2 Lifting and Substitution

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

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

where

$$\begin{aligned} \uparrow_\tau n k (TVar\ i) &= (if\ i < k\ then\ TVar\ i\ else\ TVar\ (i + n)) \\ | \uparrow_\tau n k\ Top &= Top \\ | \uparrow_\tau n k (T \rightarrow U) &= \uparrow_\tau n k\ T \rightarrow \uparrow_\tau n k\ U \\ | \uparrow_\tau n k (\forall <: T. U) &= (\forall <: \uparrow_\tau n k\ T. \uparrow_\tau n (k + 1)\ U) \end{aligned}$$

primrec $lift :: nat \Rightarrow nat \Rightarrow trm \Rightarrow trm (\uparrow)$

where

$$\begin{aligned} \uparrow n k (Var\ i) &= (if\ i < k\ then\ Var\ i\ else\ Var\ (i + n)) \\ | \uparrow n k (\lambda:T. t) &= (\lambda:\uparrow_\tau n k\ T. \uparrow n (k + 1)\ t) \\ | \uparrow n k (\lambda<:T. t) &= (\lambda<:\uparrow_\tau n k\ T. \uparrow n (k + 1)\ t) \\ | \uparrow n k (s \cdot t) &= \uparrow n k\ s \cdot \uparrow n k\ t \\ | \uparrow n k (t \cdot_\tau T) &= \uparrow n k\ t \cdot_\tau \uparrow_\tau n k\ T \end{aligned}$$

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

$$\begin{aligned} (TVar\ i)[k \mapsto_\tau S]_\tau &= \\ & (if\ k < i\ then\ TVar\ (i - 1)\ else\ if\ i = k\ then\ \uparrow_\tau k\ 0\ S\ else\ TVar\ i) \\ | Top[k \mapsto_\tau S]_\tau &= Top \\ | (T \rightarrow U)[k \mapsto_\tau S]_\tau &= T[k \mapsto_\tau S]_\tau \rightarrow U[k \mapsto_\tau S]_\tau \\ | (\forall <: T. U)[k \mapsto_\tau S]_\tau &= (\forall <: T[k \mapsto_\tau S]_\tau. U[k+1 \mapsto_\tau S]_\tau) \end{aligned}$$

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

where

$$\begin{aligned} \downarrow_\tau 0 k T &= T \\ | \downarrow_\tau (Suc\ n) k T &= \downarrow_\tau n k (T[k \mapsto_\tau Top]_\tau) \end{aligned}$$

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

where

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

$$| (\lambda <: T. t)[k \mapsto s] = (\lambda <: \downarrow_\tau 1 k T. t[k+1 \mapsto s])$$

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

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

Lifting and substitution extends to typing contexts as follows:

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

$$\begin{aligned} \uparrow_e n k \ [] &= [] \\ | \uparrow_e n k (B :: \Gamma) &= \text{mapB } (\uparrow_\tau n (k + \|\Gamma\|)) B :: \uparrow_e n k \Gamma \end{aligned}$$

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

$$\begin{aligned} \|\ [k \mapsto_\tau T]_e &= \|\ \\ | (B :: \Gamma)[k \mapsto_\tau T]_e &= \text{mapB } (\lambda U. U[k + \|\Gamma\| \mapsto_\tau T]_\tau) B :: \Gamma[k \mapsto_\tau T]_e \end{aligned}$$

primrec *decE* :: *nat* \Rightarrow *nat* \Rightarrow *env* \Rightarrow *env* (\downarrow_e)
where

$$\begin{aligned} \downarrow_e 0 k \Gamma &= \Gamma \\ | \downarrow_e (Suc n) k \Gamma &= \downarrow_e n k (\Gamma[k \mapsto_\tau \text{Top}]_e) \end{aligned}$$

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*]: $\|\uparrow_e n k \Gamma\| = \|\Gamma\|$
 $\langle \text{proof} \rangle$

lemma *substE-length* [*simp*]: $\|\Gamma[k \mapsto_\tau U]_e\| = \|\Gamma\|$
 $\langle \text{proof} \rangle$

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

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

lemma *liftT-liftT* [simp]:

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

<proof>

lemma *liftT-liftT'* [simp]:

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

<proof>

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

<proof>

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

<proof>

lemma *lift0* [simp]: $\bigwedge i. \uparrow 0 i t = t$

<proof>

theorem *substT-liftT* [simp]:

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

<proof>

theorem *liftT-substT* [simp]:

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

<proof>

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

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

<proof>

lemma *liftT-substT-Top* [simp]:

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

<proof>

lemma *liftT-substT-strange*:

$$\uparrow_\tau n k T[n + k \mapsto_\tau U]_\tau = \uparrow_\tau n (Suc k) T[k \mapsto_\tau \uparrow_\tau n 0 U]_\tau$$

<proof>

lemma *lift-lift* [simp]:

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

<proof>

lemma *substT-substT*:

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

<proof>

2.3 Well-formedness

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

inductive

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

where

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

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

inductive

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

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

where

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

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

inductive-cases *well-formed-cases*:

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

inductive-cases *well-formedE-cases*:

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

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

<proof>

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

<proof>

lemma *map-is-TVarb*:

$map\ is\text{-}TVarB\ \Gamma' = map\ is\text{-}TVarB\ \Gamma \Longrightarrow$
 $\Gamma \langle i \rangle = [TVarB\ T] \Longrightarrow \exists T. \Gamma' \langle i \rangle = [TVarB\ T]$

<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 $map\ is-TVarB\ \Gamma' = map\ is-TVarB\ \Gamma \implies \Gamma' \vdash_{wf} T$ *<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*:

$\Delta @ B :: \Gamma \vdash_{wf} \implies \Gamma \vdash_{wfB} B' \implies is-TVarB\ B' = is-TVarB\ B \implies$

$\Delta @ B' :: \Gamma \vdash_{wf}$

<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 $\uparrow_e (Suc\ 0)\ 0\ \Delta @ B :: \Gamma \vdash_{wf} \uparrow_\tau (Suc\ 0)\ \|\Delta\|\ T$

<proof>

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

<proof>

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

<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 $\Gamma\langle i \rangle = [VarB\ T] \implies \Gamma \vdash_{wf} \uparrow_\tau (Suc\ i)\ 0\ T$

<proof>

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

theorem *wf-subst*:

$$\Delta @ B :: \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow \Delta[0 \mapsto_{\tau} U]_e @ \Gamma \vdash_{wf} T[\|\Delta\| \mapsto_{\tau} U]_{\tau}$$

<proof>

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

<proof>

2.4 Subtyping

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

inductive

$$subtyping :: env \Rightarrow type \Rightarrow type \Rightarrow bool \quad (- \vdash - <: - [50, 50, 50] 50)$$

where

$$\begin{aligned} & SA-Top: \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} S \Longrightarrow \Gamma \vdash S <: Top \\ | & SA-refl-TVar: \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} TVar\ i \Longrightarrow \Gamma \vdash TVar\ i <: TVar\ i \\ | & SA-trans-TVar: \Gamma \langle i \rangle = \lfloor TVarB\ U \rfloor \Longrightarrow \\ & \quad \Gamma \vdash \uparrow_{\tau} (Suc\ i)\ 0\ U <: T \Longrightarrow \Gamma \vdash TVar\ i <: T \\ | & SA-arrow: \Gamma \vdash T_1 <: S_1 \Longrightarrow \Gamma \vdash S_2 <: T_2 \Longrightarrow \Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2 \\ | & SA-all: \Gamma \vdash T_1 <: S_1 \Longrightarrow TVarB\ T_1 :: \Gamma \vdash S_2 <: T_2 \Longrightarrow \\ & \quad \Gamma \vdash (\forall <: S_1. S_2) <: (\forall <: T_1. T_2) \end{aligned}$$

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*:

$$\text{assumes } PQ: \Gamma \vdash P <: Q$$

shows $\Gamma \vdash_{wf} \langle proof \rangle$

lemma *wf-subtype*:

$$\text{assumes } PQ: \Gamma \vdash P <: Q$$

shows $\Gamma \vdash_{wf} P \wedge \Gamma \vdash_{wf} Q \langle proof \rangle$

lemma *wf-subtypeE*:

$$\text{assumes } H: \Gamma \vdash T <: U$$

$$\text{and } H': \Gamma \vdash_{wf} \Longrightarrow \Gamma \vdash_{wf} T \Longrightarrow \Gamma \vdash_{wf} U \Longrightarrow P$$

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

$\Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} T \leq T$
<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 $H: \Delta @ \Gamma \vdash P \leq Q$
and $wf: \Gamma \vdash_{wf} B$
shows $\uparrow_e 1 0 \Delta @ B :: \Gamma \vdash \uparrow_\tau 1 \|\Delta\| P \leq \uparrow_\tau 1 \|\Delta\| Q$ *<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

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

lemma *subtype-trans*: — A.3

$$\begin{aligned} \Gamma \vdash S <: Q &\Longrightarrow \Gamma \vdash Q <: T \Longrightarrow \Gamma \vdash S <: T \\ \Delta @ TVarB Q :: \Gamma \vdash M <: N &\Longrightarrow \Gamma \vdash P <: Q \Longrightarrow \\ \Delta @ TVarB P :: \Gamma \vdash M <: N & \\ \langle proof \rangle \end{aligned}$$

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

lemma *substT-subtype*: — A.10

assumes $H: \Delta @ TVarB Q :: \Gamma \vdash S <: T$
shows $\Gamma \vdash P <: Q \Longrightarrow \Delta[0 \mapsto_\tau P]_e @ \Gamma \vdash S[||\Delta|| \mapsto_\tau P]_\tau <: T[||\Delta|| \mapsto_\tau P]_\tau$
 $\langle proof \rangle$

lemma *subst-subtype*:

assumes $H: \Delta @ VarB V :: \Gamma \vdash T <: U$
shows $\downarrow_e 1 0 \Delta @ \Gamma \vdash \downarrow_\tau 1 ||\Delta|| T <: \downarrow_\tau 1 ||\Delta|| U$
 $\langle proof \rangle$

2.5 Typing

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

inductive

typing :: *env* \Rightarrow *trm* \Rightarrow *type* \Rightarrow *bool* ($- \vdash - : -$ [50, 50, 50] 50)

where

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

Note that in the rule *T-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 $\Gamma \vdash_{wf} \langle proof \rangle$

theorem *wf-typeE2*:

assumes $H: \Gamma \vdash t : T$
shows $\Gamma \vdash_{wf} T \langle proof \rangle$

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*: $\Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wf} \wedge \Gamma \vdash_{wf} T$
 $\langle proof \rangle$

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

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

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 $\Gamma \vdash T <: U \rightarrow U' \Longrightarrow$
 $(\bigwedge S'. \Gamma \vdash U <: S \Longrightarrow VarB S :: \Gamma \vdash s : S' \Longrightarrow$
 $\Gamma \vdash \downarrow_{\tau} 1 0 S' <: U' \Longrightarrow P) \Longrightarrow P$
 $\langle proof \rangle$

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

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

lemma *TAbs-type'*:
assumes $H: \Gamma \vdash (\lambda<:S. s) : (\forall<:U. U')$
and $R: \bigwedge S'. \Gamma \vdash U <: S \Longrightarrow TVarB U :: \Gamma \vdash s : S' \Longrightarrow$
 $TVarB U :: \Gamma \vdash S' <: U' \Longrightarrow P$
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 $\Gamma \vdash_{wfB} B \implies$
 $\uparrow_e 1 0 \Delta @ B :: \Gamma \vdash \uparrow 1 \|\Delta\| t : \uparrow_\tau 1 \|\Delta\| T \langle proof \rangle$

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

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 $H: \Delta @ VarB U :: \Gamma \vdash t : T$
shows $\Gamma \vdash u : U \implies$
 $\downarrow_e 1 0 \Delta @ \Gamma \vdash t[\|\Delta\| \mapsto u] : \downarrow_\tau 1 \|\Delta\| T \langle proof \rangle$

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

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 definition of the evaluation relation $t \mapsto t'$. 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

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

where

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

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

theorem *preservation*: — A.20

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

shows $t \mapsto t' \Longrightarrow \Gamma \vdash t' : T$ *<proof>*

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

lemma *Fun-canonical*: — A.14(1)

assumes $ty: \square \vdash v : T_1 \rightarrow T_2$

shows $v \in value \Longrightarrow \exists t S. v = (\lambda:S. t)$ *<proof>*

lemma *TyAll-canonical*: — A.14(3)

assumes $ty: \square \vdash v : (\forall <: T_1. T_2)$

shows $v \in value \Longrightarrow \exists t S. v = (\lambda <: S. t)$ *<proof>*

theorem *progress*:

assumes $ty: \square \vdash t : T$

shows $t \in value \vee (\exists t'. t \mapsto t')$ *<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.

types $name = string$

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

types

$fldT = name\ \times\ type$
 $rcdT = (name\ \times\ type)\ list$

datatype $binding = VarB\ type\ | TVarB\ type$

types $env = binding\ list$

primrec $is-TVarB :: binding\ \Rightarrow\ bool$

where

$is-TVarB\ (VarB\ T) = False$
 $| is-TVarB\ (TVarB\ T) = True$

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

where

$type-ofB\ (VarB\ T) = T$
 $| type-ofB\ (TVarB\ T) = T$

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

where

$mapB\ f\ (VarB\ T) = VarB\ (f\ T)$
 $| mapB\ f\ (TVarB\ T) = TVarB\ (f\ T)$

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* (($\exists \lambda$:-./ -) [0, 10] 10)
| *TAbs type trm* (($\exists \lambda$ <:-./ -) [0, 10] 10)
| *App trm trm* (**infixl** · 200)
| *TApp trm type* (**infixl** · _{τ} 200)
| *Rcd (name × trm) list*
| *Proj trm name* ((-.-) [90, 91] 90)
| *LET pat trm trm* ((LET (- =/ -)/ IN (-)) 10)

types

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

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

LET PRcd [(*l*₁, *PVar T*₁), ..., (*l*_{*n*}, *PVar T*_{*n*})] = *Rcd* [(*l*₁, *v*₁), ..., (*l*_{*n*}, *v*_{*n*})] *IN t*

can be treated like a nested abstraction ($\lambda:T_1. \dots \lambda:T_n. t$) · *v*₁ · ... · *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* (↑ _{τ})

and *lift_rT* :: *nat* ⇒ *nat* ⇒ *rcdT* ⇒ *rcdT* (↑ _{$r\tau$})

and *lift_fT* :: *nat* ⇒ *nat* ⇒ *fldT* ⇒ *fldT* (↑ _{$f\tau$})

where

↑ _{τ} *n k* (*TVar i*) = (if *i* < *k* then *TVar i* else *TVar (i + n)*)
| ↑ _{τ} *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 *lift_p* :: *nat* \Rightarrow *nat* \Rightarrow *pat* \Rightarrow *pat* (\uparrow_p)
and *lift_{rp}* :: *nat* \Rightarrow *nat* \Rightarrow *rpat* \Rightarrow *rpat* (\uparrow_{rp})
and *lift_{fp}* :: *nat* \Rightarrow *nat* \Rightarrow *fpat* \Rightarrow *fpat* (\uparrow_{fp})

where

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

primrec *lift* :: *nat* \Rightarrow *nat* \Rightarrow *trm* \Rightarrow *trm* (\uparrow)
and *lift_r* :: *nat* \Rightarrow *nat* \Rightarrow *rcd* \Rightarrow *rcd* (\uparrow_r)
and *lift_f* :: *nat* \Rightarrow *nat* \Rightarrow *fld* \Rightarrow *fld* (\uparrow_f)

where

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

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

where

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

primrec *subst_pT* :: *pat* \Rightarrow *nat* \Rightarrow *type* \Rightarrow *pat* ($[- \mapsto_{\tau} -]_p [300, 0, 0] 300$)
and *subst_{rp}T* :: *rpat* \Rightarrow *nat* \Rightarrow *type* \Rightarrow *rpat* ($[- \mapsto_{\tau} -]_{rp} [300, 0, 0] 300$)

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

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

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

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

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

primrec $substT :: trm \Rightarrow nat \Rightarrow type \Rightarrow trm \ (-[- \mapsto_{\tau} -] [300, 0, 0] 300)$
and $substrT :: rcd \Rightarrow nat \Rightarrow type \Rightarrow rcd \ (-[- \mapsto_{\tau} -]_r [300, 0, 0] 300)$
and $substfT :: fld \Rightarrow nat \Rightarrow type \Rightarrow fld \ (-[- \mapsto_{\tau} -]_f [300, 0, 0] 300)$
where
 $(Var\ i)[k \mapsto_{\tau} S] = (if\ k < i\ then\ Var\ (i - 1)\ else\ Var\ i)$
 $| (t \cdot u)[k \mapsto_{\tau} S] = t[k \mapsto_{\tau} S] \cdot u[k \mapsto_{\tau} S]$
 $| (t \cdot_{\tau} T)[k \mapsto_{\tau} S] = t[k \mapsto_{\tau} S] \cdot_{\tau} T[k \mapsto_{\tau} S]_{\tau}$

$| (\lambda:T. t)[k \mapsto_\tau S] = (\lambda:T[k \mapsto_\tau S]_\tau. t[k+1 \mapsto_\tau S])$
 $| (\lambda<:T. t)[k \mapsto_\tau S] = (\lambda<:T[k \mapsto_\tau S]_\tau. t[k+1 \mapsto_\tau S])$
 $| (Rcd\ fs)[k \mapsto_\tau S] = Rcd\ (fs[k \mapsto_\tau S]_r)$
 $| (t..a)[k \mapsto_\tau S] = (t[k \mapsto_\tau S])..a$
 $| (LET\ p = t\ IN\ u)[k \mapsto_\tau S] =$
 $\quad (LET\ p[k \mapsto_\tau S]_p = t[k \mapsto_\tau S]\ IN\ u[k + \|p\|_p \mapsto_\tau S])$
 $| [][k \mapsto_\tau S]_r = []$
 $| (f :: fs)[k \mapsto_\tau S]_r = f[k \mapsto_\tau S]_f :: fs[k \mapsto_\tau S]_r$
 $| (l, t)[k \mapsto_\tau S]_f = (l, t[k \mapsto_\tau S])$

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

where

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

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

where

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

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

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

where

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

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

where

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

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

where

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

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

where

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

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

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

<proof>

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

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

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

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

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

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

$$\begin{aligned} k \leq k' &\implies k' < k + n \implies (\uparrow_{\tau} n k T)[k' \mapsto_{\tau} U]_{\tau} = \uparrow_{\tau} (n - 1) k T \\ k \leq k' &\implies k' < k + n \implies (\uparrow_{r\tau} n k rT)[k' \mapsto_{\tau} U]_{r\tau} = \uparrow_{r\tau} (n - 1) k rT \\ k \leq k' &\implies k' < k + n \implies (\uparrow_{f\tau} n k fT)[k' \mapsto_{\tau} U]_{f\tau} = \uparrow_{f\tau} (n - 1) k fT \\ &\langle \text{proof} \rangle \end{aligned}$$

theorem *liftT-substT* [simp]:

$$\begin{aligned} k \leq k' &\implies \uparrow_{\tau} n k (T[k' \mapsto_{\tau} U]_{\tau}) = \uparrow_{\tau} n k T[k' + n \mapsto_{\tau} U]_{\tau} \\ k \leq k' &\implies \uparrow_{r\tau} n k (rT[k' \mapsto_{\tau} U]_{r\tau}) = \uparrow_{r\tau} n k rT[k' + n \mapsto_{\tau} U]_{r\tau} \\ k \leq k' &\implies \uparrow_{f\tau} n k (fT[k' \mapsto_{\tau} U]_{f\tau}) = \uparrow_{f\tau} n k fT[k' + n \mapsto_{\tau} U]_{f\tau} \\ &\langle \text{proof} \rangle \end{aligned}$$

theorem *liftT-substT'* [simp]:

$$\begin{aligned} k' < k &\implies \\ &\uparrow_{\tau} n k (T[k' \mapsto_{\tau} U]_{\tau}) = \uparrow_{\tau} n (k + 1) T[k' \mapsto_{\tau} \uparrow_{\tau} n (k - k') U]_{\tau} \\ k' < k &\implies \\ &\uparrow_{r\tau} n k (rT[k' \mapsto_{\tau} U]_{r\tau}) = \uparrow_{r\tau} n (k + 1) rT[k' \mapsto_{\tau} \uparrow_{\tau} n (k - k') U]_{r\tau} \\ k' < k &\implies \\ &\uparrow_{f\tau} n k (fT[k' \mapsto_{\tau} U]_{f\tau}) = \uparrow_{f\tau} n (k + 1) fT[k' \mapsto_{\tau} \uparrow_{\tau} n (k - k') U]_{f\tau} \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *liftT-substT-Top* [simp]:

$$\begin{aligned} k \leq k' &\implies \uparrow_{\tau} n k' (T[k \mapsto_{\tau} Top]_{\tau}) = \uparrow_{\tau} n (Suc k') T[k \mapsto_{\tau} Top]_{\tau} \\ k \leq k' &\implies \uparrow_{r\tau} n k' (rT[k \mapsto_{\tau} Top]_{r\tau}) = \uparrow_{r\tau} n (Suc k') rT[k \mapsto_{\tau} Top]_{r\tau} \\ k \leq k' &\implies \uparrow_{f\tau} n k' (fT[k \mapsto_{\tau} Top]_{f\tau}) = \uparrow_{f\tau} n (Suc k') fT[k \mapsto_{\tau} Top]_{f\tau} \\ &\langle \text{proof} \rangle \end{aligned}$$

theorem *liftE-substE* [simp]:

$$k \leq k' \implies \uparrow_e n k (\Gamma[k' \mapsto_{\tau} U]_e) = \uparrow_e n k \Gamma[k' + n \mapsto_{\tau} U]_e$$

<proof>

lemma *liftT-decT* [simp]:

$$k \leq k' \implies \uparrow_{\tau} n k' (\downarrow_{\tau} m k T) = \downarrow_{\tau} m k (\uparrow_{\tau} n (m + k') T)$$

<proof>

lemma *liftT-substT-strange*:

$$\begin{aligned} \uparrow_{\tau} n k T[n + k \mapsto_{\tau} U]_{\tau} &= \uparrow_{\tau} n (Suc k) T[k \mapsto_{\tau} \uparrow_{\tau} n 0 U]_{\tau} \\ \uparrow_{r\tau} n k rT[n + k \mapsto_{\tau} U]_{r\tau} &= \uparrow_{r\tau} n (Suc k) rT[k \mapsto_{\tau} \uparrow_{\tau} n 0 U]_{r\tau} \\ \uparrow_{f\tau} n k fT[n + k \mapsto_{\tau} U]_{f\tau} &= \uparrow_{f\tau} n (Suc k) fT[k \mapsto_{\tau} \uparrow_{\tau} n 0 U]_{f\tau} \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *liftp-liftp* [simp]:

$$\begin{aligned} k \leq k' &\implies k' \leq k + n \implies \uparrow_p n' k' (\uparrow_p n k p) = \uparrow_p (n + n') k p \\ k \leq k' &\implies k' \leq k + n \implies \uparrow_{rp} n' k' (\uparrow_{rp} n k rp) = \uparrow_{rp} (n + n') k rp \\ k \leq k' &\implies k' \leq k + n \implies \uparrow_{fp} n' k' (\uparrow_{fp} n k fp) = \uparrow_{fp} (n + n') k fp \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *liftp-psize*[simp]:

$$\|\uparrow_p n k p\|_p = \|p\|_p$$

$$\begin{aligned} \|\uparrow_{rp} n k fs\|_r &= \|fs\|_r \\ \|\uparrow_{fp} n k f\|_f &= \|f\|_f \\ \langle proof \rangle \end{aligned}$$

lemma *lift-lift* [simp]:

$$\begin{aligned} k \leq k' &\implies k' \leq k + n \implies \uparrow n' k' (\uparrow n k t) = \uparrow (n + n') k t \\ k \leq k' &\implies k' \leq k + n \implies \uparrow_r n' k' (\uparrow_r n k fs) = \uparrow_r (n + n') k fs \\ k \leq k' &\implies k' \leq k + n \implies \uparrow_f n' k' (\uparrow_f n k f) = \uparrow_f (n + n') k f \\ \langle proof \rangle \end{aligned}$$

lemma *liftE-liftE* [simp]:

$$k \leq k' \implies k' \leq k + n \implies \uparrow_e n' k' (\uparrow_e n k \Gamma) = \uparrow_e (n + n') k \Gamma \\ \langle proof \rangle$$

lemma *liftE-liftE'* [simp]:

$$i + m \leq j \implies \uparrow_e n j (\uparrow_e m i \Gamma) = \uparrow_e m i (\uparrow_e n (j - m) \Gamma) \\ \langle proof \rangle$$

lemma *substT-substT*:

$$\begin{aligned} i \leq j &\implies \\ T[Suc j \mapsto_\tau V]_\tau [i \mapsto_\tau U [j - i \mapsto_\tau V]_\tau]_\tau &= T[i \mapsto_\tau U]_\tau [j \mapsto_\tau V]_\tau \\ i \leq j &\implies \\ rT[Suc j \mapsto_\tau V]_{r\tau} [i \mapsto_\tau U [j - i \mapsto_\tau V]_\tau]_{r\tau} &= rT[i \mapsto_\tau U]_{r\tau} [j \mapsto_\tau V]_{r\tau} \\ i \leq j &\implies \\ fT[Suc j \mapsto_\tau V]_{f\tau} [i \mapsto_\tau U [j - i \mapsto_\tau V]_\tau]_{f\tau} &= fT[i \mapsto_\tau U]_{f\tau} [j \mapsto_\tau V]_{f\tau} \\ \langle proof \rangle \end{aligned}$$

lemma *substT-decT* [simp]:

$$k \leq j \implies (\downarrow_\tau i k T) [j \mapsto_\tau U]_\tau = \downarrow_\tau i k (T [i + j \mapsto_\tau U]_\tau) \\ \langle proof \rangle$$

lemma *substT-decT'* [simp]:

$$i \leq j \implies \downarrow_\tau k (Suc j) T [i \mapsto_\tau Top]_\tau = \downarrow_\tau k j (T [i \mapsto_\tau Top]_\tau) \\ \langle proof \rangle$$

lemma *substE-substE*:

$$i \leq j \implies \Gamma [Suc j \mapsto_\tau V]_e [i \mapsto_\tau U [j - i \mapsto_\tau V]_\tau]_e = \Gamma [i \mapsto_\tau U]_e [j \mapsto_\tau V]_e \\ \langle proof \rangle$$

lemma *substT-decE* [simp]:

$$i \leq j \implies \downarrow_e k (Suc j) \Gamma [i \mapsto_\tau Top]_e = \downarrow_e k j (\Gamma [i \mapsto_\tau Top]_e) \\ \langle proof \rangle$$

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

$\langle proof \rangle$

lemma *substE-app* [simp]:

$$(\Gamma @ \Delta) [k \mapsto_\tau T]_e = \Gamma [k + \|\Delta\| \mapsto_\tau T]_e @ \Delta [k \mapsto_\tau T]_e \\ \langle proof \rangle$$

lemma *subst-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]:
 $\downarrow_e n k (B :: \Gamma) = \text{map}B (\downarrow_\tau n (k + \|\Gamma\|)) B :: \downarrow_e n k \Gamma$
 ⟨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]:
 $k \leq k' \implies k' + m \leq k + n \implies \downarrow_\tau m k' (\uparrow_\tau n k \Gamma) = \uparrow_\tau (n - m) k \Gamma$
 ⟨proof⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where

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

inductive

well-formedE :: env ⇒ bool (- ⊢_{wf} [50] 50)

and *well-formedB* :: env ⇒ binding ⇒ bool (- ⊢_{wfB} - [50, 50] 50)

where

$\Gamma \vdash_{wfB} B \equiv \Gamma \vdash_{wf} \text{type-of} B B$

| *wf-Nil*: $\square \vdash_{wf}$

| *wf-Cons*: $\Gamma \vdash_{wfB} B \implies \Gamma \vdash_{wf} \implies B :: \Gamma \vdash_{wf}$

inductive-cases *well-formed-cases*:

$\Gamma \vdash_{wf} TVar\ i$

$\Gamma \vdash_{wf} Top$

$\Gamma \vdash_{wf} T \rightarrow U$

$\Gamma \vdash_{wf} (\forall <: T. U)$

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

inductive-cases *well-formedE-cases*:

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

lemma *wf-TVarB*: $\Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} \implies TVarB\ T :: \Gamma \vdash_{wf}$

<proof>

lemma *wf-VarB*: $\Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} \implies VarB\ T :: \Gamma \vdash_{wf}$

<proof>

lemma *map-is-TVarb*:

map is-TVarB $\Gamma' = \text{map is-TVarB } \Gamma \implies$

$\Gamma \langle i \rangle = \lfloor TVarB\ T \rfloor \implies \exists T. \Gamma' \langle i \rangle = \lfloor TVarB\ T \rfloor$

<proof>

lemma *wf-equallength*:

assumes $H: \Gamma \vdash_{wf} T$

shows *map is-TVarB* $\Gamma' = \text{map is-TVarB } \Gamma \implies \Gamma' \vdash_{wf} T$ *<proof>*

lemma *wfE-replace*:

$\Delta @ B :: \Gamma \vdash_{wf} \implies \Gamma \vdash_{wfB} B' \implies is-TVarB\ B' = is-TVarB\ B \implies$

$\Delta @ B' :: \Gamma \vdash_{wf}$

<proof>

lemma *wf-weaken*:

assumes $H: \Delta @ \Gamma \vdash_{wf} T$

shows $\uparrow_e (Suc\ 0)\ 0\ \Delta @ B :: \Gamma \vdash_{wf} \uparrow_\tau (Suc\ 0) \parallel \Delta \parallel T$

<proof>

lemma *wf-weaken'*: $\Gamma \vdash_{wf} T \implies \Delta @ \Gamma \vdash_{wf} \uparrow_\tau \parallel \Delta \parallel 0\ T$

<proof>

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

<proof>

lemma *wf-liftB*:

assumes $H: \Gamma \vdash_{wf}$

shows $\Gamma \langle i \rangle = \lfloor \text{VarB } T \rfloor \implies \Gamma \vdash_{wf} \uparrow_{\tau} (\text{Suc } i) \ 0 \ T$

<proof>

theorem *wf-subst*:

$\Delta @ B :: \Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} U \implies \Delta[0 \mapsto_{\tau} U]_e @ \Gamma \vdash_{wf} T[\|\Delta\| \mapsto_{\tau} U]_{\tau}$

$\forall (l, T) \in \text{set } (rT::\text{rcdT}). \Delta @ B :: \Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} U \implies$

$\forall (l, T) \in \text{set } rT. \Delta[0 \mapsto_{\tau} U]_e @ \Gamma \vdash_{wf} T[\|\Delta\| \mapsto_{\tau} U]_{\tau}$

$\Delta @ B :: \Gamma \vdash_{wf} \text{snd } (fT::\text{fldT}) \implies \Gamma \vdash_{wf} U \implies$

$\Delta[0 \mapsto_{\tau} U]_e @ \Gamma \vdash_{wf} \text{snd } fT[\|\Delta\| \mapsto_{\tau} U]_{\tau}$

<proof>

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

<proof>

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

<proof>

3.4 Subtyping

The definition of the subtyping judgement is extended with a rule *SA-Rcd* stating that a record type $\text{RcdT } fs$ is a subtype of $\text{RcdT } fs'$, 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 $\text{RcdT } fs$ is well-formed. In order to ensure that the type $\text{RcdT } fs'$ is well-formed, too, we only have to require that labels in fs' are unique, since, by induction on the subtyping derivation, all types contained in fs' are already well-formed.

inductive

subtyping :: *env* \Rightarrow *type* \Rightarrow *type* \Rightarrow *bool* $(- \vdash - <: - [50, 50, 50] \ 50)$

where

SA-Top: $\Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} S \implies \Gamma \vdash S <: \text{Top}$

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

| *SA-trans-TVar*: $\Gamma \langle i \rangle = \lfloor \text{TVarB } U \rfloor \implies$

$\Gamma \vdash \uparrow_{\tau} (\text{Suc } i) \ 0 \ U <: T \implies \Gamma \vdash \text{TVar } i <: T$

| *SA-arrow*: $\Gamma \vdash T_1 <: S_1 \implies \Gamma \vdash S_2 <: T_2 \implies \Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2$

| *SA-all*: $\Gamma \vdash T_1 <: S_1 \implies \text{TVarB } T_1 :: \Gamma \vdash S_2 <: T_2 \implies$

$\Gamma \vdash (\forall <: S_1. S_2) <: (\forall <: T_1. T_2)$

| *SA-Rcd*: $\Gamma \vdash_{wf} \implies \Gamma \vdash_{wf} \text{RcdT } fs \implies \text{unique } fs' \implies$

$\forall (l, T) \in \text{set } fs'. \exists S. (l, S) \in \text{set } fs \wedge \Gamma \vdash S <: T \implies \Gamma \vdash \text{RcdT } fs <: \text{RcdT } fs'$

lemma *wf-subtype-env*:

assumes $PQ: \Gamma \vdash P <: Q$
shows $\Gamma \vdash_{wf} \langle proof \rangle$

lemma *wf-subtype*:

assumes $PQ: \Gamma \vdash P <: Q$
shows $\Gamma \vdash_{wf} P \wedge \Gamma \vdash_{wf} Q \langle proof \rangle$

lemma *wf-subtypeE*:

assumes $H: \Gamma \vdash T <: U$
and $H': \Gamma \vdash_{wf} P \implies \Gamma \vdash_{wf} T \implies \Gamma \vdash_{wf} U \implies P$
shows P
 $\langle proof \rangle$

lemma *subtype-refl*: — A.1

$\Gamma \vdash_{wf} P \implies \Gamma \vdash_{wf} P \implies \Gamma \vdash P <: P$
 $\Gamma \vdash_{wf} P \implies \forall (l::name, T) \in set fTs. \Gamma \vdash_{wf} T \longrightarrow \Gamma \vdash T <: T$
 $\Gamma \vdash_{wf} P \implies \Gamma \vdash_{wf} snd (fT::fldT) \implies \Gamma \vdash snd fT <: snd fT$
 $\langle proof \rangle$

lemma *subtype-weaken*:

assumes $H: \Delta @ \Gamma \vdash P <: Q$
and $wf: \Gamma \vdash_{wf} B$
shows $\uparrow_e 1 0 \Delta @ B :: \Gamma \vdash \uparrow_\tau 1 \|\Delta\| P <: \uparrow_\tau 1 \|\Delta\| Q \langle proof \rangle$

lemma *subtype-weaken'*: — A.2

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

lemma *fieldT-size [simp]*:

$(a, T) \in set fs \implies size T < Suc (list-size (prod-size (list-size char-size) size) fs)$
 $\langle proof \rangle$

lemma *subtype-trans*: — A.3

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

lemma *substT-subtype*: — A.10

assumes $H: \Delta @ TVarB Q :: \Gamma \vdash S <: T$
shows $\Gamma \vdash P <: Q \implies$
 $\Delta[0 \mapsto_\tau P]_e @ \Gamma \vdash S[\|\Delta\| \mapsto_\tau P]_\tau <: T[\|\Delta\| \mapsto_\tau P]_\tau$
 $\langle proof \rangle$

lemma *subst-subtype*:

assumes $H: \Delta @ VarB V :: \Gamma \vdash T <: U$

shows $\downarrow_e 1 0 \Delta @ \Gamma \vdash \downarrow_\tau 1 \|\Delta\| T <: \downarrow_\tau 1 \|\Delta\| U$
 ⟨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* \Rightarrow *type* \Rightarrow *env* \Rightarrow *bool* ($\vdash - : - \Rightarrow -$ [50, 50, 50] 50)
and *ptypings* :: *rpat* \Rightarrow *rcdT* \Rightarrow *env* \Rightarrow *bool* ($\vdash - [:] - \Rightarrow -$ [50, 50, 50] 50)

where

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

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

inductive

typing :: *env* \Rightarrow *trm* \Rightarrow *type* \Rightarrow *bool* ($\vdash - : -$ [50, 50, 50] 50)
and *typings* :: *env* \Rightarrow *rcd* \Rightarrow *rcdT* \Rightarrow *bool* ($\vdash - [:] -$ [50, 50, 50] 50)

where

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

theorem *wf-typeE1*:

$$\begin{aligned} \Gamma \vdash t : T &\Longrightarrow \Gamma \vdash_{wf} T \\ \Gamma \vdash fs \text{ [:] } fTs &\Longrightarrow \Gamma \vdash_{wf} fTs \\ \langle proof \rangle \end{aligned}$$

theorem *wf-typeE2*:

$$\begin{aligned} \Gamma \vdash t : T &\Longrightarrow \Gamma \vdash_{wf} T \\ \Gamma' \vdash fs \text{ [:] } fTs &\Longrightarrow (\forall (l, T) \in \text{set } fTs. \Gamma' \vdash_{wf} T) \wedge \\ &\quad \text{unique } fTs \wedge (\forall l. (fs\langle l \rangle? = \perp) = (fTs\langle l \rangle? = \perp)) \\ \langle proof \rangle \end{aligned}$$

lemmas *ptyping-induct = ptyping-ptypings.inducts(1)*

$$\langle of \text{ - - - } \lambda x y z. \text{True, simplified True-simps, standard, consumes 1, case-names P-Var P-Rcd} \rangle$$

lemmas *ptypings-induct = ptyping-ptypings.inducts(2)*

$$\langle of \text{ - - } \lambda x y z. \text{True, simplified True-simps, standard, consumes 1, case-names P-Nil P-Cons} \rangle$$

lemmas *typing-induct = typing-typings.inducts(1)*

$$\langle of \text{ - - - } \lambda x y z. \text{True, simplified True-simps, standard, consumes 1, case-names T-Var T-Abs T-App T-TAbs T-TApp T-Sub T-Let T-Rcd T-Proj} \rangle$$

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

$$\langle of \text{ - - } \lambda x y z. \text{True, simplified True-simps, standard, consumes 1, case-names T-Nil T-Cons} \rangle$$

lemma *narrow-type*: — A.7

$$\begin{aligned} \Delta @ TVarB Q :: \Gamma \vdash t : T &\Longrightarrow \\ \Gamma \vdash P <: Q &\Longrightarrow \Delta @ TVarB P :: \Gamma \vdash t : T \\ \Delta @ TVarB Q :: \Gamma \vdash ts \text{ [:] } Ts &\Longrightarrow \\ \Gamma \vdash P <: Q &\Longrightarrow \Delta @ TVarB P :: \Gamma \vdash ts \text{ [:] } Ts \\ \langle proof \rangle \end{aligned}$$

lemma *typings-setD*:

$$\begin{aligned} \text{assumes } H: \Gamma \vdash fs \text{ [:] } fTs \\ \text{shows } (l, T) \in \text{set } fTs &\Longrightarrow \exists t. fs\langle l \rangle? = [t] \wedge \Gamma \vdash t : T \\ \langle proof \rangle \end{aligned}$$

lemma *subtype-refl'*:

$$\begin{aligned} \text{assumes } t: \Gamma \vdash t : T \\ \text{shows } \Gamma \vdash T <: T \\ \langle proof \rangle \end{aligned}$$

lemma *Abs-type*: — A.13(1)

$$\begin{aligned} \text{assumes } H: \Gamma \vdash (\lambda:S. s) : T \\ \text{shows } \Gamma \vdash T <: U \rightarrow U' &\Longrightarrow \\ (\wedge S'. \Gamma \vdash U <: S &\Longrightarrow VarB S :: \Gamma \vdash s : S' &\Longrightarrow \end{aligned}$$

$\Gamma \vdash \downarrow_{\tau} 1 \ 0 \ S' <: U' \implies P \implies P$
 $\langle \text{proof} \rangle$

lemma *Abs-type'*:

assumes $H: \Gamma \vdash (\lambda:S. s) : U \rightarrow U'$
and $R: \bigwedge S'. \Gamma \vdash U <: S \implies \text{VarB } S :: \Gamma \vdash s : S' \implies$
 $\Gamma \vdash \downarrow_{\tau} 1 \ 0 \ S' <: U' \implies P$
shows $P \langle \text{proof} \rangle$

lemma *TAbs-type*: — A.13(2)

assumes $H: \Gamma \vdash (\lambda<:S. s) : T$
shows $\Gamma \vdash T <: (\forall <:U. U') \implies$
 $(\bigwedge S'. \Gamma \vdash U <: S \implies \text{TVarB } U :: \Gamma \vdash s : S' \implies$
 $\text{TVarB } U :: \Gamma \vdash S' <: U' \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *TAbs-type'*:

assumes $H: \Gamma \vdash (\lambda<:S. s) : (\forall <:U. U')$
and $R: \bigwedge S'. \Gamma \vdash U <: S \implies \text{TVarB } U :: \Gamma \vdash s : S' \implies$
 $\text{TVarB } U :: \Gamma \vdash S' <: U' \implies P$
shows $P \langle \text{proof} \rangle$

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

lemma *Rcd-type1*: — A.13(3)

assumes $H: \Gamma \vdash t : T$
shows $t = \text{Rcd } fs \implies \Gamma \vdash T <: \text{RcdT } fTs \implies$
 $\forall (l, U) \in \text{set } fTs. \exists u. fs\langle l \rangle? = [u] \wedge \Gamma \vdash u : U$
 $\langle \text{proof} \rangle$

lemma *Rcd-type1'*:

assumes $H: \Gamma \vdash \text{Rcd } fs : \text{RcdT } fTs$
shows $\forall (l, U) \in \text{set } fTs. \exists u. fs\langle l \rangle? = [u] \wedge \Gamma \vdash u : U$
 $\langle \text{proof} \rangle$

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

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

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

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

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

lemma *Rcd-type2*:

$$\begin{aligned} \Gamma \vdash \text{Rcd } fs : T &\Longrightarrow \Gamma \vdash T <: \text{Rcd} T \text{ } fTs \Longrightarrow \\ \Gamma \vdash \text{map } (\lambda(l, T). (l, \text{the } (fs\langle l \rangle?))) fTs &[:] fTs \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *Rcd-type2'*:

$$\begin{aligned} \text{assumes } H: \Gamma \vdash \text{Rcd } fs : \text{Rcd} T \text{ } fTs \\ \text{shows } \Gamma \vdash \text{map } (\lambda(l, T). (l, \text{the } (fs\langle l \rangle?))) fTs &[:] fTs \\ \langle \text{proof} \rangle \end{aligned}$$

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

lemma *ptyping-length [simp]*:

$$\begin{aligned} \vdash p : T \Rightarrow \Delta \Longrightarrow \|p\|_p = \|\Delta\| \\ \vdash fps [:] fTs \Rightarrow \Delta \Longrightarrow \|fps\|_r = \|\Delta\| \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *lift-ptyping*:

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

lemma *type-weaken*:

$$\begin{aligned} \Delta @ \Gamma \vdash t : T \Longrightarrow \Gamma \vdash_{wfB} B \Longrightarrow \\ \uparrow_e 1 0 \Delta @ B :: \Gamma \vdash \uparrow 1 \|\Delta\| t : \uparrow_\tau 1 \|\Delta\| T \\ \Delta @ \Gamma \vdash fs [:] fTs \Longrightarrow \Gamma \vdash_{wfB} B \Longrightarrow \\ \uparrow_e 1 0 \Delta @ B :: \Gamma \vdash \uparrow_r 1 \|\Delta\| fs [:] \uparrow_{r\tau} 1 \|\Delta\| fTs \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *type-weaken'*: — A.5(6)

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

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*:

$$\begin{aligned} \vdash p : T \Rightarrow \Delta \Longrightarrow \vdash p[k \mapsto_\tau U]_p : T[k \mapsto_\tau U]_\tau \Rightarrow \Delta[k \mapsto_\tau U]_e \\ \vdash fps [:] fTs \Rightarrow \Delta \Longrightarrow \vdash fps[k \mapsto_\tau U]_{rp} [:] fTs[k \mapsto_\tau U]_{r\tau} \Rightarrow \Delta[k \mapsto_\tau U]_e \\ \langle \text{proof} \rangle \end{aligned}$$

theorem *subst-type*: — A.8

$$\begin{aligned} \Delta @ \text{Var} B U :: \Gamma \vdash t : T \Longrightarrow \Gamma \vdash u : U \Longrightarrow \\ \downarrow_e 1 0 \Delta @ \Gamma \vdash t[\|\Delta\| \mapsto u] : \downarrow_\tau 1 \|\Delta\| T \\ \Delta @ \text{Var} B U :: \Gamma \vdash fs [:] fTs \Longrightarrow \Gamma \vdash u : U \Longrightarrow \\ \downarrow_e 1 0 \Delta @ \Gamma \vdash fs[\|\Delta\| \mapsto u]_r [:] \downarrow_{r\tau} 1 \|\Delta\| fTs \\ \langle \text{proof} \rangle \end{aligned}$$

theorem *substT-type*: — A.11

$$\Delta @ \text{TVar} B Q :: \Gamma \vdash t : T \Longrightarrow \Gamma \vdash P <: Q \Longrightarrow$$

$$\begin{aligned}
& \Delta[0 \mapsto_{\tau} P]_e @ \Gamma \vdash t[\|\Delta\| \mapsto_{\tau} P] : T[\|\Delta\| \mapsto_{\tau} P]_{\tau} \\
& \Delta @ TVarB Q :: \Gamma \vdash fs[:] fTs \Longrightarrow \Gamma \vdash P <: Q \Longrightarrow \\
& \Delta[0 \mapsto_{\tau} P]_e @ \Gamma \vdash fs[\|\Delta\| \mapsto_{\tau} P]_r[:] fTs[\|\Delta\| \mapsto_{\tau} P]_{r\tau} \\
& \langle proof \rangle
\end{aligned}$$

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

inductive

$match :: pat \Rightarrow trm \Rightarrow trm \ list \Rightarrow bool \ (\vdash - \triangleright - \Rightarrow - [50, 50, 50] 50)$
 $\text{and } matches :: rpat \Rightarrow rcd \Rightarrow trm \ list \Rightarrow bool \ (\vdash - [\triangleright] - \Rightarrow - [50, 50, 50] 50)$
where
 $M-PVar: \vdash PVar T \triangleright t \Rightarrow [t]$
 $| M-Rcd: \vdash fps [\triangleright] fs \Rightarrow ts \Longrightarrow \vdash PRcd fps \triangleright Rcd fs \Rightarrow ts$
 $| M-Nil: \vdash [] [\triangleright] fs \Rightarrow []$
 $| M-Cons: fs \langle l \rangle? = [t] \Longrightarrow \vdash p \triangleright t \Rightarrow ts \Longrightarrow \vdash fps [\triangleright] fs \Rightarrow us \Longrightarrow$
 $\quad \vdash (l, p) :: fps [\triangleright] fs \Rightarrow ts @ us$

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

inductive

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

| *E-hd*: $t \mapsto t' \implies (l, t) :: fs \ [\mapsto] (l, t') :: fs$
| *E-tl*: $v \in \text{value} \implies fs \ [\mapsto] fs' \implies (l, v) :: fs \ [\mapsto] (l, v) :: fs'$

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

lemmas *matchs-induct* = *match-matchs.inducts(2)*

[of - - $\lambda x y z. \text{True}$, *simplified True-simps*, *standard*, *consumes 1*,
case-names M-Nil M-Cons]

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

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

lemma *matchs-mono*:

assumes $H: \vdash fps \ [\triangleright] fs \Rightarrow ts$

shows $fps\langle l \rangle? = \perp \implies \vdash fps \ [\triangleright] (l, t) :: fs \Rightarrow ts$

<proof>

lemma *matchs-eq*:

assumes $H: \vdash fps \ [\triangleright] fs \Rightarrow ts$

shows $\forall (l, p) \in \text{set } fps. fs\langle l \rangle? = fs'\langle l \rangle? \implies \vdash fps \ [\triangleright] fs' \Rightarrow ts$

<proof>

lemma *reorder-eq*:

assumes $H: \vdash fps \ [:] fTs \Rightarrow \Delta$

shows $\forall (l, U) \in \text{set } fTs. \exists u. fs\langle l \rangle? = [u] \implies$

$\forall (l, p) \in \text{set } fps. fs\langle l \rangle? = (\text{map } (\lambda(l, T). (l, \text{the } (fs\langle l \rangle?))) fTs)\langle l \rangle?$

<proof>

lemma *matchs-reorder*:

$\vdash fps \ [:] fTs \Rightarrow \Delta \implies \forall (l, U) \in \text{set } fTs. \exists u. fs\langle l \rangle? = [u] \implies$

$\vdash fps \ [\triangleright] fs \Rightarrow ts \implies \vdash fps \ [\triangleright] \text{map } (\lambda(l, T). (l, \text{the } (fs\langle l \rangle?))) fTs \Rightarrow ts$

<proof>

lemma *matchs-reorder'*:

$\vdash fps \ [:] fTs \Rightarrow \Delta \implies \forall (l, U) \in \text{set } fTs. \exists u. fs\langle l \rangle? = [u] \implies$

$\vdash fps \ [\triangleright] \text{map } (\lambda(l, T). (l, \text{the } (fs\langle l \rangle?))) fTs \Rightarrow ts \implies \vdash fps \ [\triangleright] fs \Rightarrow ts$

<proof>

theorem *matchs-tl*:

assumes $H: \vdash fps \ [\triangleright] (l, t) :: fs \Rightarrow ts$

shows $fps\langle l \rangle? = \perp \implies \vdash fps \ [\triangleright] fs \Rightarrow ts$

<proof>

theorem *match-length*:

$\vdash p \triangleright t \Rightarrow ts \implies \vdash p : T \Rightarrow \Delta \implies \|ts\| = \|\Delta\|$

$$\vdash fps [\triangleright] ft \Rightarrow ts \Longrightarrow \vdash fps [:] fTs \Rightarrow \Delta \Longrightarrow \|ts\| = \|\Delta\|$$

<proof>

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

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

<proof>

lemma *evals-labels* [*simp*]:

$$\begin{aligned} & \text{assumes } H: fs \mapsto fs' \\ & \text{shows } (fs \langle l \rangle? = \perp) = (fs' \langle l \rangle? = \perp) \text{ } \langle proof \rangle \end{aligned}$$

theorem *preservation*: — A.20

$$\begin{aligned} & \Gamma \vdash t : T \Longrightarrow t \mapsto t' \Longrightarrow \Gamma \vdash t' : T \\ & \Gamma \vdash fs [:] fTs \Longrightarrow fs \mapsto fs' \Longrightarrow \Gamma \vdash fs' [:] fTs \end{aligned}$$

<proof>

lemma *Fun-canonical*: — A.14(1)

$$\begin{aligned} & \text{assumes } ty: [] \vdash v : T_1 \rightarrow T_2 \\ & \text{shows } v \in \text{value} \Longrightarrow \exists t \ S. v = (\lambda:S. t) \text{ } \langle proof \rangle \end{aligned}$$

lemma *TyAll-canonical*: — A.14(3)

$$\begin{aligned} & \text{assumes } ty: [] \vdash v : (\forall <: T_1. T_2) \\ & \text{shows } v \in \text{value} \Longrightarrow \exists t \ S. v = (\lambda <: S. t) \text{ } \langle proof \rangle \end{aligned}$$

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

lemma *RcdT-canonical*: — A.14(2)

$$\begin{aligned} & \text{assumes } ty: [] \vdash v : \text{RcdT } fTs \\ & \text{shows } v \in \text{value} \Longrightarrow \\ & \quad \exists fs. v = \text{Rcd } fs \wedge (\forall (l, t) \in \text{set } fs. t \in \text{value}) \text{ } \langle proof \rangle \end{aligned}$$

theorem *reorder-prop*:

$$\begin{aligned} & \forall (l, t) \in \text{set } fs. P \ t \Longrightarrow \forall (l, U) \in \text{set } fTs. \exists u. fs \langle l \rangle? = [u] \Longrightarrow \\ & \quad \forall (l, t) \in \text{set } (\text{map } (\lambda(l, T). (l, \text{the } (fs \langle l \rangle?))) \ fTs). P \ t \end{aligned}$$

<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*:

$$\begin{aligned} & \vdash p : T \Rightarrow \Delta \Longrightarrow \boxed{\vdash} t : T \Longrightarrow t \in \text{value} \Longrightarrow \\ & \quad \exists ts. \vdash p \triangleright t \Rightarrow ts \\ & \vdash fps \text{ [:] } fTs \Rightarrow \Delta \Longrightarrow \boxed{\vdash} fs \text{ [:] } fTs \Longrightarrow \\ & \quad \forall (l, t) \in \text{set } fs. t \in \text{value} \Longrightarrow \exists us. \vdash fps \text{ [\triangleright] } fs \Rightarrow us \\ & \langle \text{proof} \rangle \end{aligned}$$

theorem *progress*: — A.16

$$\begin{aligned} & \boxed{\vdash} t : T \Longrightarrow t \in \text{value} \vee (\exists t'. t \longmapsto t') \\ & \boxed{\vdash} fs \text{ [:] } fTs \Longrightarrow (\forall (l, t) \in \text{set } fs. t \in \text{value}) \vee (\exists fs'. fs \longmapsto fs') \\ & \langle \text{proof} \rangle \end{aligned}$$

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

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

lemmas *rctxt-induct* = *ctxt-rctxt.inducts*(2)

[*of* - λx . *True*, *simplified True-simps*, *standard*, *consumes 1*,
case-names C-hd C-tl]

lemma *rctxt-labels*:

assumes $H: E \in \text{rctxt}$

shows $E t\langle l \rangle? = \perp \implies E t'\langle l \rangle? = \perp$ *<proof>*

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

inductive

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

where

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

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

lemma *context-typing*: — A.18

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

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 $\vdash p \triangleright t_1 \Rightarrow ts \implies \Gamma \vdash t_2[0 \mapsto_s ts] : T$
<proof>

theorem *Proj-preservation*: — A.19(4)

assumes $H: \Gamma \vdash Rcd\ fs..l : T$
shows $fs\langle l \rangle? = [v] \implies \Gamma \vdash v : T$

<proof>

theorem preservation: — A.20

assumes $H: t \mapsto_c t'$

shows $\Gamma \vdash t : T \implies \Gamma \vdash t' : T$ *<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

$\square \vdash t : T \implies$

$t \in \text{value} \vee (\exists E t_0 t_0'. E \in \text{ctxt} \wedge t = E t_0 \wedge t_0 \mapsto_c t_0')$

$\square \vdash fs [:] fTs \implies$

$(\forall (l, t) \in \text{set fs}. t \in \text{value}) \vee (\exists E t_0 t_0'. E \in \text{rctxt} \wedge fs = E t_0 \wedge t_0 \mapsto_c t_0')$

<proof>

theorem progress: — A.16

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

shows $t \in \text{value} \vee (\exists t'. t \mapsto_c t')$

<proof>

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

lemma ctxt-imp-eval:

$E \in \text{ctxt} \implies t \mapsto t' \implies E t \mapsto E t'$

$E_r \in \text{rctxt} \implies t \mapsto t' \implies E_r t [\mapsto] E_r t'$

<proof>

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

<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 \mapsto , we introduce the inductive predicate $t \Downarrow u$, denoting that u is

a normal form of t .

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

where

$t \in value \Longrightarrow t \Downarrow t$
 $| t \mapsto s \Longrightarrow s \Downarrow u \Longrightarrow t \Downarrow u$

definition *normal-forms* **where**

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

lemma [*code-ind-set*]: $Rcd [] \in value$

<proof>

lemma [*code-ind-set*]: $t \in value \Longrightarrow Rcd fs \in value \Longrightarrow Rcd ((l, t) \# fs) \in value$

<proof>

lemmas [*code-ind-set*] = $value.Abs\ value.TAbs$

definition

natT :: *type* **where**

$natT \equiv \forall <: Top. (\forall <: TVar\ 0. (\forall <: TVar\ 1. (TVar\ 2 \rightarrow TVar\ 1) \rightarrow TVar\ 0 \rightarrow TVar\ 1))$

definition

fact2 :: *trm* **where**

fact2 \equiv

$LET\ PVar\ natT =$
 $(\lambda <: Top. \lambda <: TVar\ 0. \lambda <: TVar\ 1. \lambda : TVar\ 2 \rightarrow TVar\ 1. \lambda : TVar\ 1. Var\ 1 \cdot$
 $Var\ 0)$
 IN
 $LET\ PRcd$
 $[(\text{"pluspp"}, PVar\ (natT \rightarrow natT \rightarrow natT)),$
 $(\text{"multpp"}, PVar\ (natT \rightarrow natT \rightarrow natT))] = Rcd$
 $[(\text{"multpp"}, \lambda : natT. \lambda : natT. \lambda <: Top. \lambda <: TVar\ 0. \lambda <: TVar\ 1. \lambda : TVar\ 2 \rightarrow$
 $TVar\ 1.$
 $Var\ 5 \cdot_{\tau} TVar\ 3 \cdot_{\tau} TVar\ 2 \cdot_{\tau} TVar\ 1 \cdot (Var\ 4 \cdot_{\tau} TVar\ 3 \cdot_{\tau} TVar\ 2 \cdot_{\tau}$
 $TVar\ 1) \cdot Var\ 0),$
 $(\text{"pluspp"}, \lambda : natT. \lambda : natT. \lambda <: Top. \lambda <: TVar\ 0. \lambda <: TVar\ 1. \lambda : TVar\ 2 \rightarrow$
 $TVar\ 1. \lambda : TVar\ 1.$
 $Var\ 6 \cdot_{\tau} TVar\ 4 \cdot_{\tau} TVar\ 3 \cdot_{\tau} TVar\ 3 \cdot Var\ 1 \cdot$
 $(Var\ 5 \cdot_{\tau} TVar\ 4 \cdot_{\tau} TVar\ 3 \cdot_{\tau} TVar\ 2 \cdot Var\ 1 \cdot Var\ 0))]$
 IN
 $Var\ 0 \cdot (Var\ 1 \cdot Var\ 2 \cdot Var\ 2) \cdot Var\ 2$

value *normal-forms fact2*

code-module *EvalF*

contains *normal-forms Set*

code-module *Test*

```

imports EvalF
contains fact2

```

$\langle ML \rangle$

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.

The same game with the predicate compiler and the generic code generator

```

lemma [code-pred-intro]: valuep (Rcd [])
   $\langle proof \rangle$ 

```

```

lemma [code-pred-intro]: valuep  $t \implies$  valuep (Rcd fs)  $\implies$  valuep (Rcd ((l, t) #
fs))
   $\langle proof \rangle$ 

```

```

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

```

```

code-pred valuep
   $\langle proof \rangle$ 

```

```

thm valuep.equation

```

```

code-pred (modes:  $i \implies o \implies$  bool as normalize) norm  $\langle proof \rangle$ 

```

```

thm norm.equation

```

```

lemma [code]: value = valuep
   $\langle proof \rangle$ 

```

```

declare mem-def[code-inline]

```

```

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.