Noname manuscript No.
(will be inserted by the editor)

A Solution to the PoplMark Challenge using de Bruijn indices in Isabelle/HOL

Stefan Berghofer

the date of receipt and acceptance should be inserted later

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. We also discuss different styles of formalizing the evaluation relation, and how this choice influences executability of the specification.

1 Introduction

Mechanisms for *binding* variables form an integral part of many programming languages, be it in the form of *parameters* of procedures or methods, function abstractions in the λ-calculus, or the *restriction* operator in the π-calculus for defining local channels. Recently, there has been an increasing interest in techniques for formalizing such calculi involving variable bindings using theorem provers. To assess the suitability of existing proof assistants for the formalization of such calculi, Aydemir et al. have issued the PoplMark Challenge [2], which has as a goal the formalization of the basic metatheory of System $F_{<:}$, including the definition of an evaluation relation and a proof of type safety. In this paper, we present a solution to this challenge in the theorem prover Isabelle/HOL [9] using de Bruijn indices. We mainly focus on the definition of the calculus and the statement of the central theorems, without discussing their proofs in detail. The interested reader may find the full proof scripts on the web[1].

The rest of the paper is structured as follows: In §3, we give a definition of the basic calculus of System $F_{<:}$ without records. In §4, we describe how the calculus introduced in §3 can be extended with records. In §5, we present an alternative definition of the

Supported by BMBF in the VerisoftXT project under grant 01 IS 07008 F

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany

[1] `http://afp.sf.net/entries/POPLmark-deBruijn.shtml`

evaluation relation using *evaluation contexts*. In §6, we discuss the executability of the specification of the evaluation relation.

2 Notation

Before we start with the actual presentation of the formalization of System $F_{<:}$, we introduce some notation that we will use in the rest of the paper. In Isabelle, a theorem with premises A, B and C, and conclusion D is denoted by

$$A \Longrightarrow B \Longrightarrow C \Longrightarrow D$$

To make statements of theorems more readable, they will often be written as

If A *and* B *and* C *then* D.

All statements written in this way have been formally proved in Isabelle, and their description has been generated automatically from their internal representation using Isabelle's document preparation system. Introduction rules of *inductive predicates*, such as typing judgements and evaluation relations, are written as

$$\frac{A \qquad B \qquad C}{D}$$

to make them correspond more closely to the notation used in textbooks about type systems and programming language semantics.

In Isabelle/HOL, lists are denoted by $[x_1, \ldots, x_n]$, where $[]$ is the empty set and $\mathrel{::}$ is the *Cons* operator. The *append* operator is written as @, and $\|xs\|$ denotes the length of list xs. In order to represent functions that may fail to return a result, such as lookup operations on lists, we use the *option* datatype with constructors $\bot$ (*None*) and $\lfloor \ldots \rfloor$ (*Some*). Using this datatype, the function $xs\langle i \rangle$ for looking up the ith element of a list can be defined as follows:

$$[]\langle i \rangle = \bot$$
$$(x \mathrel{::} xs)\langle i \rangle = (\mathit{case}\ i\ \mathit{of}\ 0 \Rightarrow \lfloor x \rfloor \mid \mathit{Suc}\ j \Rightarrow xs\langle j \rangle)$$

Function *the* returns x when applied to $\lfloor x \rfloor$.

3 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*.

3.1 Types and Terms

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

datatype *type* = *TVar nat* | *Top* | *Fun type type* | *TyAll type type*

We use $T \to U$ and $\forall <: T.\ U$ as syntactic sugar for *Fun T U* and *TyAll T U*, respectively. The following datatype represents the terms of System $F_{<:}$:

datatype *trm* =
  Var nat | *Abs type trm* | *TAbs type trm* | *App trm trm* | *TApp trm type*

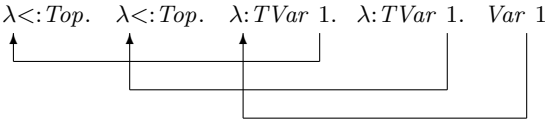
To improve readability, we write $\lambda : T.\ t$ and $\lambda <: T.\ t$ instead of *Abs T t* and *TAbs T t*, as well as $t \cdot u$ and $t \cdot_\tau T$ instead of *App t u* and *TApp t T*, respectively.

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, represented by the datatype

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

where the ith element $\Gamma\langle i \rangle$ corresponds to the variable with index i. 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. It should be noted that this is a purely syntactic issue due to our use of Isabelle's built-in list type. 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.

To better understand how contexts work, it is useful to have a look at an example. Assume we would like to type check the (polymorphic) K combinator, which in textbook notation would be written as $\lambda A <: Top.\ \lambda B <: Top.\ \lambda x : A.\ \lambda y : B.\ x$. In order to show that this term has type $\forall A <: Top.\ \forall B <: Top.\ A \to B \to A$, it suffices to show that x has type A in the context $A <: Top,\ \forall B <: Top,\ x : A,\ y : B$. More formally, we have to show that the typing judgement $A <: Top,\ \forall B <: Top,\ x : A,\ y : B \vdash x : A$ holds. Using de Bruijn syntax, the K combinator becomes



Note that both the variable A in the abstraction $\lambda x : A$ and the variable B in the abstraction $\lambda x : B$ are represented by the index 1, since in both cases there is one abstraction between the variable and its binding abstraction. The same holds for the representation of the variable x in the body of the innermost abstraction. The same set of indices is used for both type and term variables, therefore it does not matter whether the abstractions that are located between a variable and its binding abstraction are abstractions over term or type variables. Showing that the de Bruijn version of the K combinator has type $\forall <: Top.\ \forall <: Top.\ TVar\ 1 \to TVar\ 0 \to TVar\ 1$ amounts to showing that the typing judgement

$[VarB\ (TVar\ 1),\ VarB\ (TVar\ 1),\ TVarB\ Top,\ TVarB\ Top] \vdash Var\ 1 : TVar\ 3$

holds. Just like a type variable in a term can only refer to type variables bound by *enclosing* abstractions, a type variable in a binding can only refer to context elements *on its right*, with 0 referring to the leftmost of these elements, rather than to arbitrary elements in the whole context.

3.2 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 function on terms is defined by

$$\uparrow^n_k (Var\ i) = (\mathsf{if}\ i < k\ \mathsf{then}\ Var\ i\ \mathsf{else}\ Var\ (i + n))$$
$$\uparrow^n_k (\lambda{:}T.\ t) = (\lambda{:}\uparrow_\tau{}^n_k\ T.\ \uparrow^n_{k+1}\ t)$$
$$\uparrow^n_k (\lambda{<:}T.\ t) = (\lambda{<:}\uparrow_\tau{}^n_k\ T.\ \uparrow^n_{k+1}\ t)$$
$$\uparrow^n_k (s \cdot t) = \uparrow^n_k\ s \cdot \uparrow^n_k\ t$$
$$\uparrow^n_k (t \cdot_\tau T) = \uparrow^n_k\ t \cdot_\tau \uparrow_\tau{}^n_k\ T$$

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

$$Var\ i[k \mapsto s] = (\mathsf{if}\ k < i\ \mathsf{then}\ Var\ (i-1)\ \mathsf{else\ if}\ i = k\ \mathsf{then}\ \uparrow^k_0\ s\ \mathsf{else}\ Var\ i)$$
$$(t \cdot u)[k \mapsto s] = t[k \mapsto s] \cdot u[k \mapsto s]$$
$$(t \cdot_\tau T)[k \mapsto s] = t[k \mapsto s] \cdot_\tau \downarrow_\tau{}^1_k\ T$$
$$(\lambda{:}T.\ t)[k \mapsto s] = (\lambda{:}\downarrow_\tau{}^1_k\ T.\ t[k+1 \mapsto s])$$
$$(\lambda{<:}T.\ t)[k \mapsto s] = (\lambda{<:}\downarrow_\tau{}^1_k\ T.\ t[k+1 \mapsto s])$$

There are several things to note about this definition of substitution. Since the substitution function will be used to specify β-reduction, it actually does not only substitute the term s for the variable i, but also decrements the indices of all other free variables by 1, to compensate for the disappearance of abstractions during β-reduction. Moreover, since the same set of indices is used for both type and term variables, the free variables in types T have to be decremented as well, to ensure that they still point to the right positions. In a calculus with dependent types, where types may depend on terms, a substitution of a term in a type would have to be used in place of the unlifting function. Of course, the substitution functions only make sense for well-formed types and well-typed terms, and if k is of the right kind, i.e. k must denote a type variable in $T[k \mapsto_\tau S]_\tau$ and $t[k \mapsto_\tau S]$, and a term variable in $t[k \mapsto s]$. For example, the substitution $Var\ i[i \mapsto_\tau S]$ does not return a meaningful result. Likewise, the unlifting function $\downarrow_\tau{}^n_k\ T$ only makes sense if the variables with indices i such that $k \leq i \leq k + n - 1$ are term variables, i.e. no type variables. Some standard properties of lifting and substitution, which can be proved by structural induction on terms and types, are given in Figure 1. Properties of this kind are quite standard for encodings using de Bruijn indices and can also be found in papers by Huet [7], Barras and Werner [3], Nipkow [8], and Rasmussen [12].

Lifting and substitution extends to typing contexts as follows:

$$\uparrow_e{}^n_k\ [] = []$$
$$\uparrow_e{}^n_k\ (B :: \Gamma) = mapB \uparrow_\tau{}^n_{k-1}\ B :: \uparrow_e{}^n_{k-1}\ \Gamma$$

$$[][k \mapsto_\tau T]_e = []$$
$$(B :: \Gamma)[k \mapsto_\tau T]_e = mapB\ (\lambda U.\ U[k-1 \mapsto_\tau T]_\tau)\ B :: \Gamma[k-1 \mapsto_\tau T]_e$$

$$i \le j \Longrightarrow j \le i + m \Longrightarrow \uparrow_\tau^n_j\ (\uparrow_\tau^m_i\ T) = \uparrow_\tau^{m\ +\ n}_i\ T$$
$$i + m \le j \Longrightarrow \uparrow_\tau^n_j\ (\uparrow_\tau^m_i\ T) = \uparrow_\tau^m_i\ (\uparrow_\tau^{n\ -\ m}_j\ T)$$
$$k \le k' \Longrightarrow k' < k + n \Longrightarrow \uparrow_\tau^n_k\ T[k' \mapsto_\tau U]_\tau = \uparrow_\tau^{n\ -\ 1}_k\ T$$
$$k \le k' \Longrightarrow \uparrow_\tau^n_k\ (T[k' \mapsto_\tau U]_\tau) = \uparrow_\tau^n_k\ T[k' + n \mapsto_\tau U]_\tau$$
$$k' < k \Longrightarrow \uparrow_\tau^n_k\ (T[k' \mapsto_\tau U]_\tau) = \uparrow_\tau^{n}_{k\ +\ 1}\ T[k' \mapsto_\tau \uparrow_\tau^n_{k\ -\ k'}\ U]_\tau$$
$$k \le k' \Longrightarrow k' \le k + n \Longrightarrow \uparrow_{k'}^{n'}\ (\uparrow_k^n\ t) = \uparrow_k^{n\ +\ n'}\ t$$
$$i \le j \Longrightarrow T[Suc\ j \mapsto_\tau V]_\tau[i \mapsto_\tau U[j - i \mapsto_\tau V]_\tau]_\tau = T[i \mapsto_\tau U]_\tau[j \mapsto_\tau V]_\tau$$

Fig. 1 Standard properties of lifting and substitution

To better understand how the lifting and substitution functions for contexts work, it is important to note that they are applied to *fragments* of contexts in the following sense: assume we have a context of the form $\Delta\ @\ \Gamma$, where $\Delta = [B_1, \ldots, B_{\|\Delta\|}]$, and we would like to insert a binding B between Δ and Γ. Since all free variables in B_i with indices j such that $j \ge \|\Delta\| - i$ refer to entries in Γ, they have to be incremented by 1 in order to still point to the right entries in Γ after the insertion of B. Free variables in B_i with indices $j < \|\Delta\| - i$ refer to entries in Δ and therefore do not have to be adjusted. The new context after insertion of B thus becomes $\uparrow_{e\ \|\Delta\|}^1\ \Delta\ @\ B :: \Gamma$. Note that the indices of free variables in Γ do not have to be adjusted, since they can only refer to entries in the context occurring further to the right and are therefore not affected by the insertion of B. Likewise, if we have a context $\Delta\ @\ TVarB\ Q :: \Gamma$ and would like to substitute the type P (where $\Gamma \vdash P <: Q$) for the type variable at position $\|\Delta\|$ in the context, we have to replace the free variable with index j in B_i by P if $j = \|\Delta\| - i$, and decrement it by 1 if $j > \|\Delta\| - i$, since the binding $TVarB\ Q$ will disappear from the context as a result of the substitution. Again, free variables in B_i with indices $j < \|\Delta\| - i$, as well as free variables in Γ do not have to be adjusted, since they are unaffected by the substitution. The new context after the application of the substitution thus becomes $\Delta[\|\Delta\| \mapsto_\tau P]_e\ @\ \Gamma$. The rationale behind the definition of lifting and substitution on contexts will become more obvious when looking at the weakening and substitution theorems in §3.5.

As already pointed out by Nipkow [8], an alternative way of defining substitution would be to use a lifting function that only increments free variables by 1. With this approach, the term s would have to be lifted whenever descending into an abstraction, but no lifting would have to take place in the variable case of the definition of substitution. The equations for the variable and abstraction cases would thus become

$$Var\ i[k \mapsto s] = (\mathsf{if}\ k < i\ \mathsf{then}\ Var\ (i - 1)\ \mathsf{else\ if}\ i = k\ \mathsf{then}\ s\ \mathsf{else}\ Var\ i)$$
$$(\lambda{:}T.\ t)[k \mapsto s] = (\lambda{:}{\downarrow_\tau}^1_k\ T.\ t[k + 1 \mapsto \uparrow_0^1\ s])$$
$$(\lambda{<:}T.\ t)[k \mapsto s] = (\lambda{<:}{\downarrow_\tau}^1_k\ T.\ t[k + 1 \mapsto \uparrow_0^1\ s])$$

While with this definition the properties to prove about substitution and lifting are slightly simpler, the definition using lifting by k is more efficient, since it avoids traversing the term s each time when an abstraction is encountered. It is therefore used in most implementations, including the core of the Isabelle proof assistant, and Pierce's reference implementation of the $F_{<:}$ type checker [11]. Our variant of substitution and lifting can be extended to contexts more smoothly, and since lifting by k is also needed in the definition of the typing rules (see Figure 4), it seemed more natural to use it throughout the development.

Types

$$\frac{}{\Gamma \vdash_{wf} \mathit{Top}} \quad (\mathit{wf\text{-}Top})$$

$$\frac{\Gamma\langle i\rangle = \lfloor \mathit{TVarB}\ T \rfloor}{\Gamma \vdash_{wf} \mathit{TVar}\ i} \quad (\mathit{wf\text{-}TVar})$$

$$\frac{\Gamma \vdash_{wf} T \qquad \Gamma \vdash_{wf} U}{\Gamma \vdash_{wf} T \to U} \quad (\mathit{wf\text{-}arrow})$$

$$\frac{\Gamma \vdash_{wf} T \qquad \mathit{TVarB}\ T :: \Gamma \vdash_{wf} U}{\Gamma \vdash_{wf} (\forall <: T.\ U)} \quad (\mathit{wf\text{-}all})$$

Contexts

$$\frac{}{[] \vdash_{wf}} \quad (\mathit{wf\text{-}Nil})$$

$$\frac{\Gamma \vdash_{wfB} B \qquad \Gamma \vdash_{wf}}{B :: \Gamma \vdash_{wf}} \quad (\mathit{wf\text{-}Cons})$$

Fig. 2 Well-formedness of types and contexts

3.3 Well-formedness

The subtyping and typing judgements to be defined in §3.4 and §3.5 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 free variables occurring in it are defined in Γ. More precisely, if T contains a free type variable $\mathit{TVar}\ i$, then the ith element of Γ must exist and have the form $\mathit{TVarB}\ U$. A context Γ is well-formed, if all types occurring in it only refer to type variables declared "further to the right". The well-formedness judgements for types and contexts, denoted by $\Gamma \vdash_{wf} T$ and $\Gamma \vdash_{wf}$, respectively, are shown in Figure 2. The judgement $\Gamma \vdash_{wfB} B$, which denotes well-formedness of the binding B with respect to context Γ, is just an abbreviation for $\Gamma \vdash_{wf} \mathit{type\text{-}ofB}\ B$. We now present a number of properties of the well-formedness judgements that will be used in the proofs in the following sections. 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:

If $\Gamma \vdash_{wf} T$ *and* $\mathit{map}\ \mathit{is\text{-}TVarB}\ \Gamma' = \mathit{map}\ \mathit{is\text{-}TVarB}\ \Gamma$ *then* $\Gamma' \vdash_{wf} T$.

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':

If $\Delta\ @\ B :: \Gamma \vdash_{wf}$ *and* $\Gamma \vdash_{wfB} B'$ *and* $\mathit{is\text{-}TVarB}\ B' = \mathit{is\text{-}TVarB}\ B$ *then* $\Delta\ @\ B' :: \Gamma \vdash_{wf}$.

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

$$\frac{\Gamma \vdash_{wf} \qquad \Gamma \vdash_{wf} S}{\Gamma \vdash S <: \mathit{Top}} \quad (\mathit{SA\text{-}Top})$$

$$\frac{\Gamma \vdash_{wf} \qquad \Gamma \vdash_{wf} \mathit{TVar}\ i}{\Gamma \vdash \mathit{TVar}\ i <: \mathit{TVar}\ i} \quad (\mathit{SA\text{-}refl\text{-}TVar})$$

$$\frac{\Gamma\langle i\rangle = \lfloor \mathit{TVarB}\ U \rfloor \qquad \Gamma \vdash \uparrow_\tau 0^{\mathit{Suc}\ i}\ U <: T}{\Gamma \vdash \mathit{TVar}\ i <: T} \quad (\mathit{SA\text{-}trans\text{-}TVar})$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \quad (\mathit{SA\text{-}arrow})$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \mathit{TVarB}\ T_1 :: \Gamma \vdash S_2 <: T_2}{\Gamma \vdash (\forall <: S_1.\ S_2) <: (\forall <: T_1.\ T_2)} \quad (\mathit{SA\text{-}all})$$

Fig. 3 Subtyping rules

If Δ @ $\Gamma \vdash_{wf} T$ *then* $\uparrow e^1_{\|\Delta\|}$ Δ @ $B :: \Gamma \vdash_{wf} \uparrow \tau^1_{\|\Delta\|}$ T.
If Δ @ $\Gamma \vdash_{wf}$ *and* $\Gamma \vdash_{wfB} B$ *then* $\uparrow e^1_{\|\Delta\|}$ Δ @ $B :: \Gamma \vdash_{wf}$.

Intuitively, the first lemma states that a type T which is well-formed in a context is still well-formed in a larger context, whereas the second lemma 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 §3.5, 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 $\mathit{Suc}\ i$:

If $\Gamma \vdash_{wf}$ *and* $\Gamma\langle i\rangle = \lfloor \mathit{VarB}\ T \rfloor$ *then* $\Gamma \vdash_{wf} \uparrow \tau 0^{\mathit{Suc}\ i}\ T$.

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

If Δ @ $B :: \Gamma \vdash_{wf} T$ *and* $\Gamma \vdash_{wf} U$ *then* $\Delta[\|\Delta\| \mapsto_\tau U]_e$ @ $\Gamma \vdash_{wf} T[\|\Delta\| \mapsto_\tau U]_\tau$.
If Δ @ $B :: \Gamma \vdash_{wf}$ *and* $\Gamma \vdash_{wf} U$ *then* $\Delta[\|\Delta\| \mapsto_\tau U]_e$ @ $\Gamma \vdash_{wf}$.

3.4 Subtyping

We now come to the definition of the subtyping judgement $\Gamma \vdash T <: U$. It is characterized by the introduction rules shown in Figure 3. 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 Γ_1 @ $B :: \Gamma_2$, where Γ_1 has the length i. Since the indices of variables in B

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.

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:

$$If\ \Gamma \vdash T <: U\ then\ \Gamma \vdash_{wf} \wedge\ \Gamma \vdash_{wf} T \wedge\ \Gamma \vdash_{wf} U.$$

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

$$If\ \Gamma \vdash_{wf}\ and\ \Gamma \vdash_{wf} T\ then\ \Gamma \vdash T <: T.$$

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, i.e.

$$If\ \Delta\ @\ \Gamma \vdash P <: Q\ and\ \Gamma \vdash_{wfB} B\ then\ \uparrow_e{}^1_{\|\Delta\|}\ \Delta\ @\ B :: \Gamma \vdash \uparrow_\tau{}^1_{\|\Delta\|}\ P <: \uparrow_\tau{}^1_{\|\Delta\|}\ Q.$$

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:

$$If\ \Gamma \vdash P <: Q\ and\ \Delta\ @\ \Gamma \vdash_{wf}\ then\ \Delta\ @\ \Gamma \vdash \uparrow_\tau{}^{\|\Delta\|}_0\ P <: \uparrow_\tau{}^{\|\Delta\|}_0\ Q.$$

An unrestricted transitivity rule has the disadvantage that it can be applied in any situation. In order to make the above definition of the subtyping relation *syntax-directed*, the transitivity rule *SA-trans-TVar* is restricted to the case where the type on the left-hand side of the <: operator is a variable. However, the unrestricted transitivity rule

$$\frac{\Gamma \vdash S <: Q \qquad \Gamma \vdash Q <: T}{\Gamma \vdash S <: T}$$

can be derived from this definition. In order for the proof to go through, we have to simultaneously prove another property called *narrowing*:

$$If\ \Delta\ @\ TVarB\ Q :: \Gamma \vdash M <: N\ and\ \Gamma \vdash P <: Q\ then$$
$$\Delta\ @\ TVarB\ P :: \Gamma \vdash M <: N.$$

The two properties are proved by nested induction. The outer induction is on the size of the type Q, whereas the two inner inductions for proving transitivity and narrowing are on the derivation of the subtyping judgements. The transitivity property is needed in the proof of narrowing, which is by induction on the derivation of $\Delta\ @\ TVarB\ Q :: \Gamma \vdash M <: N$. In the case corresponding to the rule *SA-trans-TVar*, we must prove $\Delta\ @\ TVarB\ P :: \Gamma \vdash TVar\ i <: T$. The only interesting case is the one where $i = \|\Delta\|$. By induction hypothesis, we know that $\Delta\ @\ TVarB\ P :: \Gamma \vdash \uparrow_\tau{}^{i\,+\,1}_0\ Q <: T$ and $(\Delta\ @\ TVarB\ Q :: \Gamma)\langle i\rangle = \lfloor TVarB\ Q \rfloor$. By assumption, we have $\Gamma \vdash P <: Q$ and hence $\Delta\ @\ TVarB\ P :: \Gamma \vdash \uparrow_\tau{}^{i\,+\,1}_0\ P <: \uparrow_\tau{}^{i\,+\,1}_0\ Q$ by weakening. Since $\uparrow_\tau{}^{i\,+\,1}_0\ Q$ has the same size as Q, we can use the transitivity property, which yields $\Delta\ @\ TVarB\ P :: \Gamma \vdash \uparrow_\tau{}^{i\,+\,1}_0\ P <: T$. The claim then follows easily by an application of *SA-trans-TVar*.

In the proof of the preservation theorem presented in §3.6, we will also need a substitution theorem, which is proved by induction on the subtyping derivation:

$$If\ \Delta\ @\ TVarB\ Q :: \Gamma \vdash S <: T\ and\ \Gamma \vdash P <: Q\ then$$
$$\Delta[\|\Delta\| \mapsto_\tau P]_e\ @\ \Gamma \vdash S[\|\Delta\| \mapsto_\tau P]_\tau <: T[\|\Delta\| \mapsto_\tau P]_\tau.$$

$$\frac{\Gamma \vdash_{wf} \qquad \Gamma\langle i\rangle = \lfloor VarB\ U \rfloor \qquad T = \uparrow_\tau^{Suc\ i}\ U}{\Gamma \vdash Var\ i : T} \quad (T\text{-}Var)$$

$$\frac{VarB\ T_1 :: \Gamma \vdash t_2 : T_2}{\Gamma \vdash (\lambda{:}T_1.\ t_2) : T_1 \to \downarrow_\tau^1\ T_2} \quad (T\text{-}Abs)$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \cdot t_2 : T_{12}} \quad (T\text{-}App)$$

$$\frac{TVarB\ T_1 :: \Gamma \vdash t_2 : T_2}{\Gamma \vdash (\lambda{<:}T_1.\ t_2) : (\forall{<:}T_1.\ T_2)} \quad (T\text{-}TAbs)$$

$$\frac{\Gamma \vdash t_1 : (\forall{<:}T_{11}.\ T_{12}) \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1 \cdot_\tau T_2 : T_{12}[0 \mapsto_\tau T_2]_\tau} \quad (T\text{-}TApp)$$

$$\frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash t : T} \quad (T\text{-}Sub)$$

Fig. 4 Type checking rules

3.5 Typing

We are now ready to give a definition of the typing judgement $\Gamma \vdash t : T$. Its introduction rules are shown in Figure 4. 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 the typing rule, the indices of all free type variables in T_2 have to be decremented by 1. Like for the subtyping judgement, we can again prove that all types and contexts involved in a typing judgement are well-formed:

If $\Gamma \vdash t : T$ *then* $\Gamma \vdash_{wf} \wedge\ \Gamma \vdash_{wf} T$.

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

If Δ @ $TVarB\ Q :: \Gamma \vdash t : T$ *and* $\Gamma \vdash P <: Q$ *then* Δ @ $TVarB\ P :: \Gamma \vdash t : T$.

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

If Δ @ $\Gamma \vdash t : T$ *and* $\Gamma \vdash_{wfB} B$ *then* $\uparrow e^1_{\|\Delta\|}\ \Delta$ @ $B :: \Gamma \vdash \uparrow^1_{\|\Delta\|}\ t : \uparrow_\tau{}^1_{\|\Delta\|}\ T$.

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

If $\Gamma \vdash t : T$ *and* Δ @ $\Gamma \vdash_{wf}$ *then* Δ @ $\Gamma \vdash \uparrow_0^{\|\Delta\|}\ t : \uparrow_\tau{}_0^{\|\Delta\|}\ T$.

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:

Canonical Values

$$\frac{}{(\lambda{<:}T.\ t) \in \mathit{value}} \qquad (\mathit{Abs})$$

$$\frac{}{(\lambda{<:}T.\ t) \in \mathit{value}} \qquad (\mathit{TAbs})$$

Evaluation

$$\frac{v_2 \in \mathit{value}}{(\lambda{:}T_{11}.\ t_{12}) \cdot v_2 \longmapsto t_{12}[0 \mapsto v_2]} \qquad (\mathit{E\text{-}Abs})$$

$$\frac{}{(\lambda{<:}T_{11}.\ t_{12}) \cdot_\tau T_2 \longmapsto t_{12}[0 \mapsto_\tau T_2]} \qquad (\mathit{E\text{-}TAbs})$$

$$\frac{t \longmapsto t'}{t \cdot u \longmapsto t' \cdot u} \qquad (\mathit{E\text{-}App_1})$$

$$\frac{v \in \mathit{value} \qquad t \longmapsto t'}{v \cdot t \longmapsto v \cdot t'} \qquad (\mathit{E\text{-}App_2})$$

$$\frac{t \longmapsto t'}{t \cdot_\tau T \longmapsto t' \cdot_\tau T} \qquad (\mathit{E\text{-}TApp})$$

Fig. 5 Canonical values and evaluation relation

If Δ @ *VarB* U :: $\Gamma \vdash t : T$ *and* $\Gamma \vdash u : U$ *then*
$\downarrow_e^1{}_{\|\Delta\|}\ \Delta$ @ $\Gamma \vdash t[\|\Delta\| \mapsto u] : \downarrow_\tau^1{}_{\|\Delta\|}\ T$.
If Δ @ *TVarB* Q :: $\Gamma \vdash t : T$ *and* $\Gamma \vdash P <: Q$ *then*
$\Delta[\|\Delta\| \mapsto_\tau P]_e$ @ $\Gamma \vdash t[\|\Delta\| \mapsto_\tau P] : T[\|\Delta\| \mapsto_\tau P]_\tau$.

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.

3.6 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. The notion of a *value* is used in the definition of the evaluation relation $t \longmapsto t'$. There are several ways of defining this evaluation relation: Aydemir et al. [2] advocate the use of *evaluation contexts* that allow the separation of the description of the "immediate" reduction rules, i.e. β-reduction, from the description of the context in which these reductions may occur. 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 §5. 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. The corresponding introduction rules are shown in Figure 5. Here, the rules *E-Abs* and *E-TAbs* describe the "immediate" reductions, whereas *E-App*$_1$, *E-App*$_2$, and *E-TApp* are additional congruence rules describing reductions in a context. The most important theorems of this section are the *preservation* theorem, stating that the reduction of a well-typed term does not change its type, and the *progress* theorem, stating that reduction of a well-typed term does not "get stuck" – in other words, every well-typed, closed term t is either a value, or there is a term t' to which t can be reduced. The preservation theorem

If $\Gamma \vdash t : T$ *and* $t \longmapsto t'$ *then* $\Gamma \vdash t' : T$.

is proved by induction on the derivation of $\Gamma \vdash t : T$, followed by a case distinction on the last rule used in the derivation of $t \longmapsto t'$. The progress theorem

If $[] \vdash t : T$ *then* $t \in value \vee (\exists\, t'.\ t \longmapsto t')$.

is also proved by induction on the derivation of $[] \vdash t : T$. In the induction steps, we need the following two lemmas about *canonical forms*

If $[] \vdash v : T_1 \to T_2$ *and* $v \in value$ *then* $\exists\, t\ S.\ v = (\lambda{:}S.\ t)$.
If $[] \vdash v : (\forall{<:}T_1.\ T_2)$ *and* $v \in value$ *then* $\exists\, t\ S.\ v = (\lambda{<:}S.\ t)$.

stating that closed values of types $T_1 \to T_2$ and $\forall{<:}T_1.\ T_2$ must be abstractions over term and type variables, respectively.

4 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.

4.1 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.

datatype *type* = *TVar nat* | *Top* | *Fun type type* | *TyAll type type*
| *RcdT* ((*string* × *type*) *list*)

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

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

datatype $trm = Var\ nat \mid Abs\ type\ trm \mid TAbs\ type\ trm \mid App\ trm\ trm$
$\mid TApp\ trm\ type \mid Rcd\ ((string \times trm)\ list) \mid Proj\ trm\ string$
$\mid LET\ pat\ trm\ trm$

For $Proj\ t\ a$ and $LET\ p\ t\ u$, we introduce the syntactic sugar $t..a$ and $LET\ p = t\ IN\ u$, respectively. In order to motivate the typing and evaluation rules for LET, it is important to note that an expression of the form

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

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

4.2 Lifting and Substitution

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 $\uparrow_k^n p$ and $T[k \mapsto_\tau S]_p$, respectively. Since the extension of the existing lifting and substitution functions to records is fairly standard, we only show the additional clauses of the function $t[k \mapsto s]$ for substituting term variables in terms:

$Rcd\ fs[k \mapsto s] = Rcd\ (fs[k \mapsto s]_r)$
$(t..a)[k \mapsto s] = t[k \mapsto s]..a$
$(LET\ p = t\ IN\ u)[k \mapsto s] = (LET \downarrow_k^1 p = t[k \mapsto s]\ IN\ u[k + \|p\|_p \mapsto s])$
$[][k \mapsto s]_r = []$
$(f :: fs)[k \mapsto s]_r = f[k \mapsto s]_f :: fs[k \mapsto s]_r$
$(l,\ t)[k \mapsto s]_f = (l,\ t[k \mapsto s])$

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. 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:

$t[k \mapsto_s []] = t$
$t[k \mapsto_s u :: us] = t[k - 1 \mapsto u][k - 1 \mapsto_s us]$

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.

4.3 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*.

$$\frac{\mathit{unique}\ \mathit{fs} \qquad \forall\,(l,\ T){\in}\mathit{set}\ \mathit{fs}.\ \Gamma \vdash_{\mathit{wf}} T}{\Gamma \vdash_{\mathit{wf}} \mathit{RcdT}\ \mathit{fs}} \quad (\mathit{wf\text{-}RcdT})$$

The *unique* predicate is characterized by the equations

$\mathit{unique}\ [] = \mathit{True}$
$\mathit{unique}\ (x :: \mathit{xs}) = (\mathit{xs}\langle \mathit{fst}\ x\rangle_? = \bot \wedge \mathit{unique}\ \mathit{xs})$

where $\mathit{xs}\langle a\rangle_?$ denotes lookup of an item with label a in the association list xs:

$[]\langle a\rangle_? = \bot$
$(x :: \mathit{xs})\langle a\rangle_? = (\mathsf{if}\ \mathit{fst}\ x = a\ \mathsf{then}\ \lfloor \mathit{snd}\ x\rfloor\ \mathsf{else}\ \mathit{xs}\langle a\rangle_?)$

4.4 Subtyping

The definition of the subtyping judgement is extended with a rule *SA-Rcd* stating that a record type $\mathit{RcdT}\ \mathit{fs}$ is a subtype of $\mathit{RcdT}\ \mathit{fs}'$, 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 $\Gamma \vdash_{\mathit{wf}}$ 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 $\mathit{RcdT}\ \mathit{fs}$ is well-formed. In order to ensure that the type $\mathit{RcdT}\ \mathit{fs}'$ 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.

$$\frac{\begin{array}{cc}\Gamma \vdash_{\mathit{wf}} & \Gamma \vdash_{\mathit{wf}} \mathit{RcdT}\ \mathit{fs} \\ \mathit{unique}\ \mathit{fs}' & \forall\,(l,\ T){\in}\mathit{set}\ \mathit{fs}'.\ \exists\, S.\ (l,\ S) \in \mathit{set}\ \mathit{fs} \wedge \Gamma \vdash S <: T\end{array}}{\Gamma \vdash \mathit{RcdT}\ \mathit{fs} <: \mathit{RcdT}\ \mathit{fs}'} \quad (\mathit{SA\text{-}Rcd})$$

It may seem that the premise $\mathit{unique}\ \mathit{fs}'$ is redundant, but this is not the case: without it, we may be able to derive $\Gamma \vdash \mathit{RcdT}\ \mathit{fs} <: \mathit{RcdT}\ \mathit{fs}'$ for some fs' containing the same entry $(l,\ T)$ twice.

4.5 Typing

The additional rules for extending the type checking judgement to records are shown in Figure 6. 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 $\vdash \mathit{ps}\ [:]\ \mathit{Ts} \Rightarrow \Delta$ 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. 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 typing judgement for patterns

Patterns

$$\frac{}{\vdash \mathit{PVar}\ T : T \Rightarrow [\mathit{VarB}\ T]} \quad (\mathit{P\text{-}Var})$$

$$\frac{\vdash \mathit{fps}\ [:]\ \mathit{fTs} \Rightarrow \Delta}{\vdash \mathit{PRcd}\ \mathit{fps} : \mathit{RcdT}\ \mathit{fTs} \Rightarrow \Delta} \quad (\mathit{P\text{-}Rcd})$$

$$\vdash []\ [:]\ [] \Rightarrow [] \quad (\mathit{P\text{-}Nil})$$

$$\frac{\vdash p : T \Rightarrow \Delta_1 \qquad \vdash \mathit{fps}\ [:]\ \mathit{fTs} \Rightarrow \Delta_2 \qquad \mathit{fps}\langle l\rangle_? = \bot}{\vdash (l,\ p) :: \mathit{fps}\ [:]\ (l,\ T) :: \mathit{fTs} \Rightarrow \uparrow_e^{\|\Delta_1\|}_{\|\Delta_2\|}\ \Delta_2\ @\ \Delta_1} \quad (\mathit{P\text{-}Cons})$$

Records

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \vdash p : T_1 \Rightarrow \Delta \qquad \Delta\ @\ \Gamma \vdash t_2 : T_2}{\Gamma \vdash (\mathit{LET}\ p = t_1\ \mathit{IN}\ t_2) : \downarrow_\tau^{\|\Delta\|}_0\ T_2} \quad (\mathit{T\text{-}Let})$$

$$\frac{\Gamma \vdash \mathit{fs}\ [:]\ \mathit{fTs}}{\Gamma \vdash \mathit{Rcd}\ \mathit{fs} : \mathit{RcdT}\ \mathit{fTs}} \quad (\mathit{T\text{-}Rcd})$$

$$\frac{\Gamma \vdash t : \mathit{RcdT}\ \mathit{fTs} \qquad \mathit{fTs}\langle l\rangle_? = \lfloor T \rfloor}{\Gamma \vdash t..l : T} \quad (\mathit{T\text{-}Proj})$$

$$\frac{\Gamma \vdash_{wf}}{\Gamma \vdash []\ [:]\ []} \quad (\mathit{T\text{-}Nil})$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash \mathit{fs}\ [:]\ \mathit{fTs} \qquad \mathit{fs}\langle l\rangle_? = \bot}{\Gamma \vdash (l,\ t) :: \mathit{fs}\ [:]\ (l,\ T) :: \mathit{fTs}} \quad (\mathit{T\text{-}Cons})$$

Fig. 6 Type checking rules for patterns and records

is used in the rule *T-Let*. The typing judgement for terms is defined simultaneously with a typing judgement $\Gamma \vdash \mathit{fs}\ [:]\ \mathit{fTs}$ for record fields.

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

$$\mathit{If}\ \Gamma \vdash \mathit{Rcd}\ \mathit{fs} : \mathit{RcdT}\ \mathit{fTs}\ \mathit{then}\ \forall (l,\ U) \in \mathit{set}\ \mathit{fTs}.\ \exists u.\ \mathit{fs}\langle l\rangle_? = \lfloor u \rfloor \wedge \Gamma \vdash u : U.$$

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

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

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

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

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

If $\Gamma \vdash Rcd\ fs : RcdT\ fTs$ *then*
$\Gamma \vdash map\ (\lambda(l,\ T).\ (l,\ the\ (fs\langle l\rangle_?)))\ fTs\ [:]\ fTs.$

The substitution lemmas are now proved by mutual induction on the derivations of the typing derivations for terms and lists of fields. For example, the substitution lemma for type variables in record fields now reads as follows:

If $\Delta\ @\ TVarB\ Q :: \Gamma \vdash fs\ [:]\ fTs$ *and* $\Gamma \vdash P <: Q$ *then*
$\Delta[\|\Delta\| \mapsto_\tau P]_e\ @\ \Gamma \vdash fs[\|\Delta\| \mapsto_\tau P]_r\ [:]\ fTs[\|\Delta\| \mapsto_\tau P]_{r\tau}.$

4.6 Evaluation

Figure 7 shows the additional rules that are necessary to extend the definiton of canonical values and the evaluation relation to records. 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. In order to formalize the evaluation rule for *LET*, we introduce another relation $\vdash p \rhd t \Rightarrow ts$ expressing that a pattern p matches a term t. The relation also yields a list of terms ts corresponding to the variables in the pattern. The relation is defined simultaneously with another relation $\vdash fps\ [\rhd]\ fs \Rightarrow ts$ for matching a list of field patterns *fps* against a list of fields *fs*. The relation $t \longmapsto t'$ is defined simultaneously with a relation $fs\ [\longmapsto]\ fs'$ for evaluating record fields. The "immediate" reductions, namely pattern matching and projection, are described by the rules *E-LetV* and *E-ProjRcd*, respectively, whereas *E-Proj*, *E-Rcd*, *E-Let*, *E-hd* and *E-tl* are congruence rules. 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

If $\vdash p : T_1 \Rightarrow \Delta$ *and* $\Gamma_2 \vdash t_1 : T_1$ *and* $\Gamma_1\ @\ \Delta\ @\ \Gamma_2 \vdash t_2 : T_2$ *and*
$\vdash p \rhd t_1 \Rightarrow ts$ *then* $\downarrow_e^{\|\Delta\|}_{\|\Gamma_1\|}\ \Gamma_1\ @\ \Gamma_2 \vdash t_2[\|ts\| + \|\Gamma_1\| \mapsto_s ts] : \downarrow_\tau^{\|\Delta\|}_{\|\Gamma_1\|}\ T_2.$

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. Another central property needed in the proof of the progress theorem is that well-typed matching is defined:

If $\vdash p : T \Rightarrow \Delta$ *and* $[] \vdash t : T$ *and* $t \in value$ *then* $\exists ts.\ \vdash p \rhd t \Rightarrow ts.$

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

If $[] \vdash v : RcdT\ fTs$ *and* $v \in value$ *then* $\exists fs.\ v = Rcd\ fs \wedge (\forall (l,\ t) \in set\ fs.\ t \in value).$

Canonical Values

$$\frac{\forall\,(l,\,t)\in set\ fs.\ t\in value}{Rcd\ fs\in value}\qquad(Rcd)$$

Pattern Matching

$$\frac{}{\vdash PVar\ T\rhd t\Rightarrow[t]}\qquad(M\text{-}PVar)$$

$$\frac{\vdash fps\ [\rhd]\ fs\Rightarrow ts}{\vdash PRcd\ fps\rhd Rcd\ fs\Rightarrow ts}\qquad(M\text{-}Rcd)$$

$$\frac{}{\vdash []\ [\rhd]\ fs\Rightarrow []}\qquad(M\text{-}Nil)$$

$$\frac{fs\langle l\rangle_?=\lfloor t\rfloor\qquad\vdash p\rhd t\Rightarrow ts\qquad\vdash fps\ [\rhd]\ fs\Rightarrow us}{\vdash(l,\,p)::fps\ [\rhd]\ fs\Rightarrow ts\ @\ us}\qquad(M\text{-}Cons)$$

Evaluation

$$\frac{v\in value\qquad\vdash p\rhd v\Rightarrow ts}{(LET\ p=v\ IN\ t)\longmapsto t[\|ts\|\mapsto_s ts]}\qquad(E\text{-}LetV)$$

$$\frac{fs\langle l\rangle_?=\lfloor v\rfloor\qquad v\in value}{Rcd\ fs..l\longmapsto v}\qquad(E\text{-}ProjRcd)$$

$$\frac{fs\ [\longmapsto]\ fs'}{Rcd\ fs\longmapsto Rcd\ fs'}\qquad(E\text{-}Rcd)$$

$$\frac{t\longmapsto t'}{(LET\ p=t\ IN\ u)\longmapsto(LET\ p=t'\ IN\ u)}\qquad(E\text{-}Let)$$

$$\frac{t\longmapsto t'}{(l,\,t)::fs\ [\longmapsto]\ (l,\,t')::fs}\qquad(E\text{-}hd)$$

$$\frac{v\in value\qquad fs\ [\longmapsto]\ fs'}{(l,\,v)::fs\ [\longmapsto]\ (l,\,v)::fs'}\qquad(E\text{-}tl)$$

Fig. 7 Canonical values and evaluation relation for records

5 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. The evaluation relation $t\longmapsto_c t'$ shown in Figure 8 is now characterized by the rule *E-Ctxt*, which allows reductions in arbitrary contexts, as well as additional rules describing the "immediate" reductions. These are exactly

Term Contexts

$$\frac{}{(\lambda t.\ t) \in ctxt} \qquad (C\text{-}Hole)$$

$$\frac{E \in ctxt}{(\lambda t.\ E\ t \cdot u) \in ctxt} \qquad (C\text{-}App_1)$$

$$\frac{v \in value \qquad E \in ctxt}{(\lambda t.\ v \cdot E\ t) \in ctxt} \qquad (C\text{-}App_2)$$

$$\frac{E \in ctxt}{(\lambda t.\ E\ t \cdot_\tau T) \in ctxt} \qquad (C\text{-}TApp)$$

$$\frac{E \in ctxt}{(\lambda t.\ E\ t..l) \in ctxt} \qquad (C\text{-}Proj)$$

$$\frac{E \in rctxt}{(\lambda t.\ Rcd\ (E\ t)) \in ctxt} \qquad (C\text{-}Rcd)$$

$$\frac{E \in ctxt}{(\lambda t.\ LET\ p = E\ t\ IN\ u) \in ctxt} \qquad (C\text{-}Let)$$

Record Contexts

$$\frac{E \in ctxt}{(\lambda t.\ (l,\ E\ t) :: fs) \in rctxt} \qquad (C\text{-}hd)$$

$$\frac{v \in value \qquad E \in rctxt}{(\lambda t.\ (l,\ v) :: E\ t) \in rctxt} \qquad (C\text{-}tl)$$

Evaluation Relation

$$\frac{t \longmapsto_c t' \qquad E \in ctxt}{E\ t \longmapsto_c E\ t'} \qquad (E\text{-}Ctxt)$$

Fig. 8 Evaluation contexts and evaluation relation using contexts

the same as the rules *E-Abs*, *E-TAbs*, *E-LetV*, and *E-ProjRcd*, which have already been presented in §3.6 and §4.6, and are therefore not shown here.

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 $E\ t'$:

If $\Gamma \vdash E\ t : T$ *and* $E \in ctxt$ *and* $\forall\, T_0.\ \Gamma \vdash t : T_0 \longrightarrow \Gamma \vdash t' : T_0$ *then* $\Gamma \vdash E\ t' : T$.

The proof is by mutual induction on the typing derivations for terms and records. We also need another lemma, which the authors of the PoplMark challenge [2] describe as follows: *If* $\Gamma \vdash t : T$ *and* $t \longmapsto_c t'$, *where the derivation* $t \longmapsto_c t'$ *ends in a rule other than E-Ctxt, then* $\Gamma \vdash t' : T$. Unfortunately, it is impossible in Isabelle to express

any constraints on the shape of a derivation like "ends in a rule other than ..." inside the logic. We therefore have to split up this lemma into several parts, one for each immediate reduction:

If $\Gamma \vdash (\lambda{:}T_{11}.\ t_{12}) \cdot t_2 : T$ *then* $\Gamma \vdash t_{12}[0 \mapsto t_2] : T$.
If $\Gamma \vdash (\lambda{<:}T_{11}.\ t_{12}) \cdot_\tau T_2 : T$ *then* $\Gamma \vdash t_{12}[0 \mapsto_\tau T_2] : T$.
If $\Gamma \vdash (LET\ p = t_1\ IN\ t_2) : T$ *and* $\vdash p \rhd t_1 \Rightarrow ts$ *then* $\Gamma \vdash t_2[\|ts\| \mapsto_s ts] : T$.
If $\Gamma \vdash Rcd\ fs..l : T$ *and* $fs\langle l\rangle_? = \lfloor v \rfloor$ *then* $\Gamma \vdash v : T$.

Each of these parts has to be proved by an induction on the typing derivation.

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.

If $[] \vdash t : T$ *then*
$t \in value \vee (\exists E\ t_0\ t_0'.\ E \in ctxt \wedge t = E\ t_0 \wedge t_0 \longmapsto_c t_0')$.

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.

We can easily prove that the definition of the evaluation relation using congruence rules is equivalent to the one using evaluation contexts:

$(t \longmapsto t') = (t \longmapsto_c t')$

Both directions of the above equality can be proved by an induction on the derivation of the evaluation relation, where the proof that $t \longmapsto_c t'$ implies $t \longmapsto t'$ requires a lemma stating that $\longmapsto$ is compatible with evaluation contexts:

If $E \in ctxt$ *and* $t \longmapsto t'$ *then* $E\ t \longmapsto E\ t'$.

This can be proved by induction on the definition of evaluation contexts.

6 Executability

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. Isabelle has a built-in code generator [6], that translates specifications into the functional programming language ML. To account for non-determinism, inductively defined predicates are translated into functions taking an input and returning a sequence (also called "lazy list") of possible outputs, encoded by type `'a seq`. For more information on how to implement lazy lists in an eager programming language such as ML, see e.g. the book by Paulson [10]. The computational behaviour of Prolog can be modeled using a *list monad*, where the "bind" operator of type `'a seq * ('a -> 'b seq) -> 'b seq` is defined by

```
fun s :-> f = Seq.flat (Seq.map f s);
```

```
fun eval x1 =
  Seq.single (x1) :->
    (fn (App (Abs (T_1_1, t_1_2), v_2)) =>
      value v_2 :-> (fn () => Seq.single (subst t_1_2 zero v_2))
      | _ => Seq.empty) ++
  Seq.single (x1) :->
    (fn (TApp (TAbs (T_1_1, t_1_2), T_2)) => Seq.single (substT t_1_2 zero T_2)
      | _ => Seq.empty) ++
  Seq.single (x1) :->
    (fn (App (t, u)) => eval t :-> (fn (t') => Seq.single (App (t', u)))
      | _ => Seq.empty) ++
  Seq.single (x1) :->
    (fn (App (v, t)) =>
      value v :->
        (fn () => eval t :-> (fn (t') => Seq.single (App (v, t'))))
      | _ => Seq.empty) ++
  Seq.single (x1) :->
    (fn (TApp (t, T)) => eval t :-> (fn (t') => Seq.single (TApp (t', T)))
      | _ => Seq.empty) ++
  Seq.single (x1) :->
    (fn (LET (p, v, t)) =>
      value v :->
        (fn () =>
          match p v :-> (fn (ts) => Seq.single (substs t zero ts)))
      | _ => Seq.empty) ++
  Seq.single (x1) :->
    (fn (Proj (Rcd fs, l)) =>
      equal (assoc fs l) :->
        (fn (Some v) => value v :-> (fn () => Seq.single (v))
          | _ => Seq.empty)
      | _ => Seq.empty) ++
  Seq.single (x1) :->
    (fn (Proj (t, l)) => eval t :-> (fn (t') => Seq.single (Proj (t', l)))
      | _ => Seq.empty) ++
  Seq.single (x1) :->
    (fn (Rcd fs) => evals fs :-> (fn (fs') => Seq.single (Rcd fs'))
      | _ => Seq.empty) ++
  Seq.single (x1) :->
    (fn (LET (p, t, u)) =>
      eval t :-> (fn (t') => Seq.single (LET (p, t', u))) | _ => Seq.empty)

and evals x1 =
  Seq.single (x1) :->
    (fn (((l, t) :: fs)) =>
      eval t :-> (fn (t') => Seq.single (((l, t') :: fs)))
      | _ => Seq.empty) ++
  Seq.single (x1) :->
    (fn (((l, v) :: fs)) =>
      value v :->
        (fn () => evals fs :-> (fn (fs') => Seq.single (((l, v) :: fs'))))
      | _ => Seq.empty);
```

Fig. 9 Code for the evaluation relation

Terms

$$\frac{}{t \rightsquigarrow \langle \lambda s.\ s,\ t\rangle} \quad (D\text{-}Hole)$$

$$\frac{t \rightsquigarrow \langle E,\ t'\rangle}{t \cdot u \rightsquigarrow \langle \lambda s.\ E\ s \cdot u,\ t'\rangle} \quad (D\text{-}App_1)$$

$$\frac{v \in value \qquad t \rightsquigarrow \langle E,\ t'\rangle}{v \cdot t \rightsquigarrow \langle \lambda s.\ v \cdot E\ s,\ t'\rangle} \quad (D\text{-}App_2)$$

$$\frac{t \rightsquigarrow \langle E,\ t'\rangle}{t \cdot_\tau T \rightsquigarrow \langle \lambda s.\ E\ s \cdot_\tau T,\ t'\rangle} \quad (D\text{-}TApp)$$

$$\frac{t \rightsquigarrow \langle E,\ t'\rangle}{t..l \rightsquigarrow \langle \lambda s.\ E\ s..l,\ t'\rangle} \quad (D\text{-}Proj)$$

$$\frac{fs\ [\rightsquigarrow]\ \langle E,\ t'\rangle}{Rcd\ fs \rightsquigarrow \langle \lambda s.\ Rcd\ (E\ s),\ t'\rangle} \quad (D\text{-}Rcd)$$

$$\frac{t \rightsquigarrow \langle E,\ t'\rangle}{(LET\ p = t\ IN\ u) \rightsquigarrow \langle \lambda s.\ LET\ p = E\ s\ IN\ u,\ t'\rangle} \quad (D\text{-}Let)$$

Records

$$\frac{t \rightsquigarrow \langle E,\ t'\rangle}{(l,\ t) :: fs\ [\rightsquigarrow]\ \langle \lambda s.\ (l,\ E\ s) :: fs,\ t'\rangle} \quad (D\text{-}hd)$$

$$\frac{v \in value \qquad fs\ [\rightsquigarrow]\ \langle E,\ t'\rangle}{(l,\ v) :: fs\ [\rightsquigarrow]\ \langle \lambda s.\ (l,\ v) :: E\ s,\ t'\rangle} \quad (D\text{-}tl)$$

Fig. 10 Decomposition of terms and records

The definition of the single-step evaluation relation presented in §3.6 and §4.6 is directly executable. Its ML code is shown in Figure 9. The `++` operator used in the code denotes *concatenation* of sequences. Function `eval` in the code shown in Figure 9, which corresponds to the evaluation relation $t \longmapsto t'$, takes as an input a term to be evaluated, and returns a sequence of possible reducts. It is defined simultaneously with a function `evals` for evaluating lists of record fields, which is used in the code for evaluating records of the form *Rcd fs*. It turns out that the evaluation relation is *deterministic*, i.e. the returned sequence will consist of at most one element. Because of this, the use of the `seq` monad is somewhat sub-optimal. Isabelle's infrastructure for generating code from inductively defined predicates is still under development [5], and future versions will include a determinism analysis to address this shortcoming. Note that the definition based on evaluation contexts from §5 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 u such that $t = E\ u$. However, from the rules defining evaluation contexts, one can easily derive a relation $t \rightsquigarrow \langle E,\ u\rangle$ that decomposes a given term into an evaluation context E, together with a term u such that $t = E\ u$. A similar relation $fs\ [\rightsquigarrow]\ \langle E,\ u\rangle$ has to be defined for lists of record fields. The introduction rules for the decomposition

relation are derived from the original rules characterizing contexts in a canonical way. The first component of the tuple $\langle \ldots, \ldots \rangle$ is the same context as the one in the conclusion of the corresponding rule for *ctxt*. The term on the left of $\rightsquigarrow$ is obtained from this context by stripping off the abstraction and replacing the subterm of the form $E\ s$ by a new variable, say t. This term t is recursively decomposed into a context E and a subterm t', the latter of which becomes the second component of the tuple $\langle \ldots, \ldots \rangle$. Correctness of the decomposition can be proved by a straightforward induction on the derivation of $t \rightsquigarrow \langle E, u \rangle$:

If $t \rightsquigarrow \langle E, u \rangle$ *then* $E \in ctxt \wedge t = E\ u$.

Similarly, completeness is proved by induction on the derivation of $E \in ctxt$:

If $E \in ctxt$ *then* $E\ u \rightsquigarrow \langle E, u \rangle$.

Using the first of these two properties, we can derive an alternative rule *E-Ctxt'*, which is executable:

$$\frac{t \rightsquigarrow \langle E, u \rangle \qquad u \longmapsto_c u'}{t \longmapsto_c E\ u'}$$

7 Conclusion

We have demonstrated that state-of-the-art proof assistants like Isabelle/HOL are quite well-suited for formalizing the meta-theory of type systems such as System $F_{<:}$. The formalization of the basic calculus without records takes up about 1300 lines of Isabelle code, the extension to records increased the formalization by another 1200 lines. Interestingly, the formalization using evaluation contexts has about the same size as the one using additional congruence rules.

The formalization of the calculus with records was obtained by copying the simpler one, and then extending the datatypes for terms and types with additional constructors. Moreover, rules for the new constructs were added to the inductive definitions of the typing, subtyping and evaluation relations. Finally, extra cases were added to the proofs. This copy-and-paste approach worked quite well for the calculus described in this article. For larger calculi, however, it would be very advantageous to have an infrastructure for supporting such incremental developments. To the best of our knowledge, this is not yet offered by any of the proof assistants used to solve the PoplMark challenge. Providing such an infrastructure is nontrivial: not only does the proof assistant have to provide inductive definitions that are extensible, but there also has to be a mechanism for checking that existing proofs are not invalidated by the extension.

Apart from our solution, only the solutions by Vouillon [15] using Coq, as well as the one by Ashley-Rollman, Crary and Harper [1] using Twelf address all parts of the PoplMark challenge. Vouillon's solution also uses de Bruijn indices to model bound variables, albeit with a slightly different encoding of contexts. While in our solution $TVar\ i$ refers to the ith binding in the context, it refers to the ith *type variable binding* in Vouillon's solution. For example, given the context $[VarB\ T_0,\ TVarB\ T_1,\ VarB\ T_2,\ TVarB\ T_3]$, the binding $TVarB\ T_3$ corresponds to $TVar\ 3$ in our solution, whereas it would correspond to $TVar\ 1$ in Vouillon's solution. Because of this,

his solution requires fewer lifting operations and can also do without the "unlifting" operation introduced in §3. In contrast, the lookup of variables and the insertion of bindings in the context, as required for example in the statement of the weakening property, seem to be a bit more complicated in Vouillon's approach. Interestingly, the reference implementation of the $F_{<:}$ type checker to be found in the book by Pierce [11] uses exactly the same encoding of contexts as our formalization, requiring "unlifting" operations in the typing rules for $\lambda{:}T.\ t$ and LET. It seems that due to the fact that the same set of indices is used for both type and term variables, our formalization is slightly easier to adapt to a dependent calculus than Vouillon's. Moreover, the encoding of *evaluation contexts* presented in §5 differs from the one chosen by Vouillon in that it uses a higher-order encoding of contexts as functions from terms to terms, rather that a first-order datatype with a specific constructor for encoding "holes". This spares us the definition of an additional function for inserting terms into the holes in a context. Although elegant, the higher-order abstract syntax encoding in Twelf cannot easily deal with binding constructs where the number of bound variables is not known beforehand, as it is the case for the LET expression. In the Twelf solution [1], this expression is encoded as `let : pattern -> term -> bterm -> term`, where `bterm` is an auxiliary datatype with constructors `base : term -> bterm` and `bind : tp -> (term -> bterm) -> bterm`. The body of `let` then has to be enclosed in as many `bind` constructors as there are variables in the `pattern`. This complication does not arise in a solution using de Bruijn indices.

Although de Bruijn indices are often criticized as being unreadable, we did not find their use particularly cumbersome. In fact, most technical lemmas about lifting and substitution could be borrowed from an existing formalization of untyped λ-calculus by Nipkow [8], as well as from a formalization of simply-typed λ-calculus [4] based on Nipkow's work. It is worth noting that we were not just able to reuse the statements of these lemmas, but also the corresponding proof scripts, which could be adapted to System $F_{<:}$ by changing just a few tactics. In contrast to other approaches such as *nominal logic* or *higher-order abstract syntax* as implemented in Twelf, which require a large amount of infrastructure [14], or rely on extensive meta-theoretic reasoning on paper [13], de Bruijn indices only need very little background theory, and are therefore ideal for getting started quickly. Moreover, they are very close to the actual implementation, which makes it easy to generate executable prototypes from the formal definition of the calculus.

References

1. M. Ashley-Rollman, K. Crary, and R. Harper. Solution to the POPLMARK Challenge in Twelf. Available electronically at `http://fling-l.seas.upenn.edu/~plclub/mmm/`, 2005.
2. 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.
3. B. Barras and B. Werner. Coq in Coq. To appear in Journal of Automated Reasoning.
4. S. Berghofer. Extracting a normalization algorithm in Isabelle/HOL. In J.-C. Filliâtre, C. Paulin, and B. Werner, editors, *Types for Proofs and Programs (TYPES 2004)*, volume 3839 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
5. S. Berghofer, L. Bulwahn, and F. Haftmann. Turning inductive into equational specifications. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *22nd International*

Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), volume 5674 of *Lecture Notes in Computer Science*, pages 131–146. Springer-Verlag, 2009.
6. S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs: TYPES'2000*, volume 2277 of *LNCS*. Springer-Verlag, 2002.
7. G. P. Huet. Residual theory in lambda-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.
8. T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66, 2001.
9. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
10. L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
11. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
12. O. Rasmussen. The Church-Rosser theorem in Isabelle: A proof porting experiment. Technical Report 364, University of Cambridge, Computer Laboratory, May 1995.
13. C. Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2000.
14. C. Urban and C. Tasson. Nominal Techniques in Isabelle/HOL. In *Proceedings of the 20th International Conference on Automated Deduction (CADE)*, volume 3632 of *LNCS*, pages 38–53, 2005.
15. J. Vouillon. Solution to the POPLMARK Challenge in Coq. Available electronically at `http://fling-l.seas.upenn.edu/~plclub/mmm/`, 2005.