Master Parisien de Recherche en Informatique 2014–2015
Proof Assistants

Coq Project
—
Stratified System F Part I

Version of January 7, 2015

Assia Mahboubi - assia.mahboubi@inria.fr
Matthieu Sozeau - matthieu.sozeau@inria.fr

The goal of this project is to model a version of predicative System F and prove that it is strongly normalizing using the technique of hereditary substitution, as in the article "Hereditary Substitution for Stratified System F" [1].

The project has two parts:

- We first define the type system and prove basic meta-theoretical results about it: strengthening, weakening, transitivity of kinding and regularity.
- In a second part (to be given end of January along with a solution for the first part), we will define a wellfounded ordering on types used to define a total hereditary substitution function which computes normal forms given two terms in normal form, one being substituted in the other. This allows to prove strong normalization of the calculus by a simple induction on typing derivations.

1 Defining the Type System

1.1 Kinds, types, terms and environments

We consider a predicative variant of (explicit) System F with a hierarchy of kinds. Kinds are identified with natural numbers and represent the levels of types. The usual order $\leq$ on natural numbers models the inclusion of a small kind in a larger one. Polymorphic quantification is kinded in this system: $\forall X : k, \tau$ models a function which takes a type $U : k$ to a term of type $\tau[U/X]$. The rest of the type language is standard: we have variables and arrow types. The term language includes λ-calculus with type abstraction and application. Variables bound in abstractions are decorated with their type, as in explicit (also known as Church-style) System F.

$$\begin{array}{llll}
(\textit{kinds}) & k, l & ::= & *_n \qquad (n \in \mathbb{N}) \\
(\textit{types}) & \tau, \mu & ::= & X \mid \tau \to \mu \mid \forall X : k, \tau \\
(\textit{terms}) & t, u, v & ::= & x \mid \lambda x : \tau . t \mid (t\ u) \mid \Lambda X : k . t \mid t[\tau]
\end{array}$$

Variables can be represented natural numbers representing de Bruijn indices (see Wikipedia[1] and Chapter 6 of [2]). In this setting, term and type variables refer to an environment, viewed as

[1]`http://en.wikipedia.org/wiki/De_Bruijn_index`

a stack: variable n denotes the variable introduced by the declaration at depth n in the environment. This is just a suggestion, it is also possible to choose the usual notation where variables are represented by an identifier or the locally nameless representation. See the POPLmark webpage[2] for examples of various techniques used to represent binders.

Note that we have two families of variables here: term variables, which are bound to typing declarations $(x : \tau)$, and type variables, which are bound to kinding declarations $(X : k)$. We recommend using a single environment type, mixing the two kinds of declarations. This way type declarations can be skipped over when looking up the kind declaration of a type variable and vice versa, resulting in two parallel sets of de Bruijn indices, e.g. the judgment

$$X : k, Y : k', x : X, y : Y, Z : k'' \vdash x : X$$

would be represented by

$$\mathsf{etvar}\ k, \mathsf{etvar}\ k', \mathsf{evar}\ (\mathsf{tvar}\ 1), \mathsf{evar}\ (\mathsf{tvar}\ 0), \mathsf{etvar}\ k'' \vdash \mathsf{var}\ 1 : \mathsf{tvar}\ 2$$

A single set of de Bruijn indices can also be used, then the encoded judgment would be:

$$\mathsf{etvar}\ k, \mathsf{etvar}\ k', \mathsf{evar}\ (\mathsf{tvar}\ 1), \mathsf{evar}\ (\mathsf{tvar}\ 1), \mathsf{etvar}\ k'' \vdash \mathsf{var}\ 2 : \mathsf{tvar}\ 4$$

or a representation with two contexts (type variable contexts being independent of term variables):

$$(\mathsf{etvar}\ k, \mathsf{etvar}\ k', \mathsf{etvar}\ k''); (\mathsf{evar}\ (\mathsf{tvar}\ 1), \mathsf{evar}\ (\mathsf{tvar}\ 0)) \vdash \mathsf{var}\ 1 : \mathsf{tvar}\ 2$$

The well-formedness, kinding and typing rules are given in figures 3, 5 and 6 of the above mentioned article [1].

1.2 Definitions

1. Define `typ`, the type representing the syntax of types and `tsubst`, the function which substitutes a type in a type for some free (type) variable X. You will need a function for shifting types in a de Bruijn encoding.

2. Define `term`, the type representing the syntax of terms, `subst_typ` which substitutes a *type* in a *term* and `subst` which substitutes a *term* in a *term*. Again, you will need shifting functions for type and term variables in terms in a de Bruijn encoding.

3. Define environments and (potentially partial) functions to get type and kind declarations from them.

4. We will now define inductive predicates `wf_env`, `kinding` and `typing` representing the well-formedness, kinding and typing judgments. E.g, typing will have type `env` → `term` → `typ` → `Prop`.

   To simplify the development, one can define the well-formedness of environments separately from the kinding and typing judgments, avoiding their mutual dependency. To do so, we define well-formed types and environments as follows:

```
Fixpoint wf_typ (e : env) (T : typ) {struct T} : Prop :=
  match T with
  | tvar X => get_kind e X = None → False
  | arrow T1 T2 => wf_typ e T1 ∧ wf_typ e T2
  | all k T2 => wf_typ (etvar e k) T2
```

[2] http://www.seas.upenn.edu/~plclub/poplmark/

```
  end.

Fixpoint wf_env (e : env) : Prop :=
  match e with
    empty => True
  | evar e T => wf_typ e T ∧ wf_env e
  | etvar e k => wf_env e
  end.
```

I.e., well-formedness only ensures that types in the environment are well-scoped, which is a sufficient condition for the kindability of each type in the environment. The typing and kinding judgments can then be defined as non-mutual inductives. (NB, one can also define `wf_typ` and `wf_env` as inductive or even boolean valued predicates as they are decidable).

5. This system has clearly decidable kind and type inference: formalize this by defining the kind and type inference functions and showing their correctness with respect to the inductive predicates `kinding` and `typing`. These functions take an environment and a term (respectively a type) and return optionally their type (respectively minimal kind).

1.3 Basic Metatheory

We are now ready to prove the first basic results about this system, culminating in the regularity lemma (Lemma 7 of [1]).

Before tackling regularity, we must prove a host of basic results showing that well-formedness, kinding and typing are closed under the various substitution operations and weakening ($\forall \Gamma, \Gamma', \Gamma'', \Gamma, \Gamma' \vdash J \Rightarrow \Gamma, \Gamma'', \Gamma' \vdash J$).

A first fundamental property of this system is cumulativity, which is necessary to prove the term substitution lemma:

Lemma 1.1 (Cumulativity)**.** $\forall \Gamma\ \tau\ k\ k', \mathit{kinding}\ \Gamma\ \tau\ k \to k \leq k' \to \mathit{kinding}\ \Gamma\ \tau\ k'$

Hint: You can use decision procedures like the `omega` tactic to solve goals involving inequalities on naturals.

1.3.1 Type substitution

1. Define an inductive predicate `insert_kind : var → env → env → Prop` such that `insert_kind X e e'` characterizes `e'` as being the extension of `e` by a kinding declaration for variable `X`. This will require shifting declarations "after" X in e in a de Bruijn encoding.

2. Show that well-formedness, kinding and typing are invariant by weakening by a kind declaration using these. A typical lemma will have the form:

```
Lemma insert_kind_wf_env :
  ∀ (X : var) (e e' : env),
  insert_kind X e e' → wf_env e → wf_env e'.
```

Finding the right induction loading in these proofs will be key, you may need in particular to reorder the quantifiers to obtain the right induction hypotheses in some cases. Also note that depending on the way variables are represented, some lifting of indices may have to be done in the derivations, for example when we use the same term in different contexts.

3. Similarly, define an inductive predicate: `env_subst : var → typ → env → env → Prop` such that `env_subst X T e e'` is inhabited when e' is e where variable X has been substituted by T. Recall that type substitution of X by U in an environment of the form $\Delta, X : *_k, \Phi$ results in the environment $\Delta, \Phi[U/X]$.

It is recommended to start showing these lemmas for kinding derivations and type weakening and substitution first and then build on this to derive the definitions for typing derivations and type substitutions.

1.3.2 Term substitution

Note that for term substitution, as the language is not dependent, one can simply implement a `remove_var : var → env → env` function that removes a particular binding to specify substitution and weakening by a typing declaration. E.g., the term substitution lemma for typing can be stated:

```
Lemma subst_preserves_typing :
  ∀ (e : env) (x : nat) (t u : term) (V W : typ),
  typing e t V →
  typing (remove_var e x) u W → get_var e x = Some W →
  typing (remove_var e x) (subst t x u) V.
```

For weakening, one can prove:

```
Lemma typing_weakening_var_ind :
  ∀ (e : env) (x : nat) (t : term) (U : typ),
  wf_env e → typing (remove_var e x) t U → typing e (shift x t) U.
```

where `(shift x t)` shifts every term variable occurrence above or at the level of `x` in `t` by 1 (in a de Bruijn encoding).

Remark. Do not consider the depths of derivations as done in [1], it is not necessary.

After this, regularity can finally be proved.

Lemma 1.2 (Regularity). *If* $\Gamma \vdash t : \tau$ *then there exists* $p \in \mathbb{N}$ *such that* $\Gamma \vdash \tau : *_p$.

Optionally, prove that the system has a narrowing property:

Lemma 1.3 (Narrowing). *If* $\Gamma, X : k, \Gamma' \vdash t : \tau$ *and* $k' \leq k$ *then* $\Gamma, X : k', \Gamma' \vdash t : \tau$

2 Reduction and Normal Terms

We can now turn to definitions needed to prove normalization.

1. We will need a notion of reduction to state the normalization theorem. Define the notion of one-step parallel reduction for terms of the language (i.e, the congruent closure of the reduction rules in figure 2 of [1]). Define reduction $\rightsquigarrow^*$ as its transitive closure.

   Hint: The `Relations` library might be useful here.

2. Show that term application, abstraction, type abstraction and type application are congruent for $\rightsquigarrow^*$.

3. We will also need a characterization of normal terms to state normalization. To define it, use two mutually inductive predicates `normal : term → Prop` and `neutral : term → Prop`.

4. Show that `normal` and `neutral` are preserved by type substitution.

In case you are stuck...

We followed here the POPLmark contribution of Jérôme Vouillon[3] which uses a single context and de Bruijn indices, you can consult it for hints but we recommend trying to formalize without it unless you are really stuck.

References

[1] Eades, H., Stump, A.: Hereditary Substitution for Stratified System F. International Workshop on Proof-Search in Type Theories, PSTT **10** (2010)

[2] Pierce, B.C.: Types and Programming Languages. MIT press (2002)

[3] http://www.seas.upenn.edu/~plclub/poplmark/vouillon.html