A locally nameless solution to the POPLmark challenge

Xavier Leroy

N° 6098

Janvier 2007

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6098--FR+ENG

A locally nameless solution to the POPLmark challenge

Xavier Leroy

Thème SYM — Systèmes symboliques
Projet Gallium

Rapport de recherche n° 6098 — Janvier 2007 — 54 pages

Abstract: The POPLmark challenge is a collective experiment intended to assess the usability of theorem provers and proof assistants in the context of fundamental research on programming languages. In this report, we present a solution to the challenge, developed with the Coq proof assistant, and using the "locally nameless" presentation of terms with binders introduced by McKinna, Pollack, Gordon, and McBride.

Key-words: POPLmark, Coq, locally nameless, alpha-conversion, binders, type systems, metatheory, system F-sub

Une solution sans noms locaux à l'expérience POPLmark

Résumé : L'expérience collective POPLmark vise à évaluer l'utilisabilité des démonstrateurs automatiques et des assistants de preuves dans le contexte de la recherche fondamentale sur les langages de programmation. Dans ce rapport, nous présentons une solution à cette expérience, développée à l'aide de l'assistant de preuves Coq, et reposant sur la présentation "sans noms locaux" des termes avec lieurs introduite par McKinna, Pollack, Gordon, et McBride.

Mots-clés : POPLmark, Coq, sans noms locaux, alpha-conversion, lieurs, systèmes de types, métathéorie, système F-sub

Chapter 1

Introduction

1.1 The POPLmark challenge

The POPLmark challenge [ABF$^+$05] is a collective experiment intended to assess the usability of theorem provers and proof assistants in the context of fundamental research on programming languages. The need for computer assistance when formalizing and proving properties of programming languages (formal semantics and type systems) is well expressed in the statement of the challenge:

> Many proofs about programming languages are long, straightforward, and tedious, with just a few interesting cases. Their complexity arises from the management of many details rather than from deep conceptual difficulties; yet small mistakes or overlooked cases can invalidate large amounts of work. These effects are amplified as languages scale: it becomes hard to keep definitions and proofs consistent, to reuse work, and to ensure tight relationships between theory and implementations. Automated proof assistants offer the hope of significantly easing these problems. However, despite much encouraging progress in recent years and the availability of several mature tools, their use is still not commonplace.

The challenge itself consists of formalizing the operational semantics and the type system of $F_{<:}$, a typed functional language featuring polymorphism and subtyping, and proving the soundness of the type system with respect to this semantics. As a guidance, the statement of the challenge provides a detailed on-paper formalization of $F_{<:}$ and a proof of type soundness written in ordinary mathematics. The challenge itself is to express these formalization and proofs in a theorem prover in such a way that the proofs can be mechanically checked.

A dozen complete or partial solutions to the POPLmark challenge were developed, using a wide variety of proof assistants (Coq, Isabelle/HOL, Twelf, ...) and of representation techniques for the terms, types and rules of the $F_{<:}$ language. These solutions can be found on the POPLmark Web site[1]. This report presents the solution that we developed. It uses the Coq proof assistant [Coq07, BC04], as well as the so-called "locally nameless" representation introduced by McKinna, Pollack, Gordon, and McBride, [MP99, Gor94, MM04]. The complete Coq development is available online [Ler07].

1.2 Locally nameless representations

Perhaps surprisingly, the main difficulty in the POPLmark challenge is not to translate the high-level, on-paper proofs into equivalent mechanized proofs. These on-paper proofs are mostly syntactic in nature and do not involve higher mathematics. Such proofs are handled well by many existing proof assistants.

What makes POPLmark difficult is the need to correctly handle ***binders***, ***bound variables*** and *alpha-conversion*. The terms and types of $F_{<:}$ contain variables such as function parameters and type variables, as well as constructs that bind these variables, such as function abstractions $\lambda x.\ x$ in terms and universal quantification $\forall X.\ X \rightarrow X$ in types. To obtain a sound theory of the language, it is necessary to treat terms and types as equal up to alpha-conversion, that is, renamings of bound variables. For instance, the two types $\forall X.\ X \rightarrow X$ and $\forall Y.\ Y \rightarrow Y$ must be treated as equal. This is difficult to achieve with today's proof assistants, because these offer poor support for quotient sets. Several representations of binders have been investigated over the last 40 years to overcome this difficulty.

Nominal representations follow the usual mathematical practice of identifying bound variables by names and quotienting terms up to alpha-conversion of bound names. This can be internalized within the logic itself, leading to nominal logics [Pit03]. Support for a nominal logic within a proof assistant is still in its infancy, but is making significant progress [UT05]. Despite these difficulties, this approach is attractive because it is the closest to usual mathematical practice, leading to statements and proofs that are very close to what we are accustomed to see in textbooks and research papers.

de Bruijn indices avoid the issue with alpha-conversion by representing variables not by names, but by position (indices) relative to the enclosing binders: v_1 is the variable bound by the first enclosing binder, v_2 by the second, etc. For instance, the term $\lambda x.\ x\ (\lambda y.\ x\ y)$ in nominal notation is represented as $\lambda.\ v_1\ (\lambda.\ v_2\ v_1)$ in de Bruijn notation. The strength of this approach is that terms have unique representations: two terms are alpha-equivalent in nominal notation if and only if they are identical in de Bruijn notation. It is therefore easy to represent and work with in a proof assistant [Hue94]. The downside of this approach is that the meaning of de Bruijn indices is very dependent on the context. Many lifting and relocation operations over indices must be included in the statements of theorems, making them unnatural and hard to read.

[1] `http://fling-l.seas.upenn.edu/~plclub/cgi-bin/poplmark/`

Higher-order abstract syntax (HOAS) uses the functions provided by the logic to represent binders. For example, we have the following representations for the term $\lambda x.x$:

Style	Representation of $\lambda x.x$
nominal	`Lambda("x", Var "x")`
de Bruijn	`Lambda(Var 1)`
HOAS	`Lambda(fun x -> x)`

(The `fun` construct denotes a function of the logic, binding a logical variable `x` that ranges over all terms.) The beauty of this approach is that alpha-conversion and substitution of bound names are handled automatically by the logic and need not be managed explicitly. The downside of HOAS is that it is not compatible with the rich logics of general-purpose proof assistants such as Coq and Isabelle/HOL and therefore can only be supported by systems such as Twelf's metatheory, which are less expressive. Also, it leads to statements of theorems that are quite different from what we write in ordinary mathematics.

In our solution to the POPLmark challenge, we use a representation for binders that is a combination of the nominal approach with de Bruijn indices. This representation is known as the "locally nameless" approach [MP99, Gor94, MM04]. It uses de Bruijn indices to identify bound variables, and names to identify free variables. For example, we have the following representations:

Style	Representation of $\lambda x.\ y\ x$
nominal	`Lambda("x", App(Var "y", Var "x"))`
de Bruijn	`Lambda(App(Var 2, Var 1))`
locally nameless	`Lambda(App(Freevar "y", Boundvar 1))`

Like de Bruijn indices, the locally nameless representation has the very nice property that alpha-equivalent terms have syntactically equal representations. Like nominal approaches, the locally nameless representation enables us to use familiar names to refer to free variables, leading to statements of theorems that are close to ordinary mathematics.

A crucial invariant in the locally nameless approach is that representations of terms never contain free de Bruijn indices. Consequently, when recursing over a de Bruijn-closed term and encountering a binder such as `Lambda`(t), it is incorrect to recurse over t, since this term can contain `Boundvar 1` as a free de Bruijn index. The correct approach is to invent a fresh name `x` to stand for the formal parameter of the lambda-abstraction and recurse over $t[0 \leftarrow$ `x`$]$, that is, the term obtained by substituting the free variable `Freevar "x"` for all occurrences of the bound variable `Boundvar 1` in t. (See section 2.2.3 for detailed explanations.) This style of definition is slightly unnatural at first, but is reasonably easy to get used to. In particular, this substitution by a fresh name materializes the so-called "Barendregt convention" for nominal terms.

1.3 Outline

The remainder of this report is a complete step-by-step presentation of our Coq development in literate programming style. All definitions, theorems and intermediate lemmas are shown in Coq syntax, interspersed with explanations. We omit the proof scripts for all lemmas but show them for the main theorems, so that the reader can get a feeling of the size and complexity of the proof scripts. The full Coq development, including scripts, is available online [Ler07].

- Chapter 2 defines the syntax of $F_{<:}$ type expressions and the subtyping relation between types, and proves three key properties of this relation: reflexivity, transitivity, and stability by substitution. It corresponds to part 1A and a fragment of part 2A of the POPLmark challenge.
- Chapter 3 defines the syntax of terms, the type system, and the dynamic semantics of $F_{<:}$. It proves the soundness of the type system with respect to the semantics. This corresponds to the remainder of part 2A of the challenge.
- Chapter 4 shows how executions of $F_{<:}$ programs can be performed (for testing purposes), both within Coq and through "extraction" (automatic generation) of Caml code for a reference $F_{<:}$ interpreter. This is part 3 of the challenge.
- Chapter 5 concludes this report by an informal assessment of the quality of our solution, and more generally of the usability of the "locally nameless" approach and of the Coq proof assistant for POPLmark-style problems.

Chapter 2

Algorithmic subtyping

This chapter corresponds to part 1A of the POPLmark challenge, namely the formal definition of a subtyping relation between types of $F_{<:}$ and the proof of basic type-theoretic properties of this relation.

We start by "importing" three modules from the Coq library of standard definitions and theorems: *Arith* (arithmetic over natural numbers), *ZArith* (arithmetic over integers) and *List* (operations over finite lists). We also import a module *extralibrary* that we developed specially, which provides additional lemmas about list membership. This module is omitted in this report, but available as part of the Web version of this development [Ler07].

```
Require Import Arith.
Require Import ZArith.
Require Import List.
Require Import extralibrary.
```

2.1 Names and swaps of names

We use names (also called atoms) to represent free variables in terms. Any infinite type with decidable equality will do. In preparation for the second part of the challenge, we attach a kind to every name: either "type" or "term", and ensure that there are infinitely many names of each kind. Concretely, we represent names by pairs of a kind and an integer (type Z).

```
Inductive name_kind : Set :=
  | TYPE: name_kind
  | TERM: name_kind.
```

```
Definition name : Set := (name_kind × Z)%type.
```

```
Definition kind (n: name) : name_kind := fst n.
```

Equality between names is decidable.

Lemma *eq_name*: $\forall$ (*n1 n2*: *name*), {*n1* = *n2*} + {*n1* $\neq$ *n2*}.

Moreover, we have the following obvious simplification rules on tests over name equality.

Lemma *eq_name_true*:
$\forall$ (*A*: *Set*) (*n*: *name*) (*a b*: *A*), (*if eq_name n n then a else b*) = *a*.

Lemma *eq_name_false*:
$\forall$ (*A*: *Set*) (*n m*: *name*) (*a b*: *A*), *n* $\neq$ *m* $\rightarrow$ (*if eq_name n m then a else b*) = *b*.

The following lemma shows that there always exists a name of the given kind that is fresh w.r.t. the given list of names, that is, distinct from all the names in this list.

Definition *find_fresh_name* (*k*: *name_kind*) (*l*: *list name*) : *name* :=
(*k*, 1 + *fold_right* (*fun* (*n*:*name*) *x* $\Rightarrow$ *Zmax* (*snd n*) *x*) 0 *l*)%*Z*.

Lemma *find_fresh_name_is_fresh*:
$\forall$ *k l*, *let n* := *find_fresh_name k l in* $\neg$*In n l* $\wedge$ *kind n* = *k*.

Lemma *fresh_name*:
$\forall$ (*k*: *name_kind*) (*l*: *list name*), $\exists$ *n*, $\neg$*In n l* $\wedge$ *kind n* = *k*.

As argued by Pitts and others, swaps (permutations of two names) are an interesting special case of renamings. We will use swaps later to prove that our definitions are equivariant, that is, insensitive to the choices of fresh identifiers.

Definition *swap* (*u v x*: *name*) : *name* :=
if eq_name x u then v else if eq_name x v then u else x.

The following lemmas are standard properties of swaps: self-inverse, injective, kind-preserving.

Lemma *swap_left*: $\forall$ *x y*, *swap x y x* = *y*.

Lemma *swap_other*: $\forall$ *x y z*, *z* $\neq$ *x* $\rightarrow$ *z* $\neq$ *y* $\rightarrow$ *swap x y z* = *z*.

Lemma *swap_inj*: $\forall$ *u v x y*, *swap u v x* = *swap u v y* $\rightarrow$ *x* = *y*.

Lemma *swap_kind*: $\forall$ *u v x*, *kind u* = *kind v* $\rightarrow$ *kind* (*swap u v x*) = *kind x*.

2.2 Types and typing environments

2.2.1 Type expressions

The syntax of type expressions is standard, except that we have two representations for variables: *Tparam* represents free type variables, identified by a name, while *Tvar* represents bound type variables, identified by their de Bruijn indices. Our de Bruijn indices start at 0. In a *Forall t1 t2* type, the variable *Tvar* 0 is bound by the *Forall* in type *t2*.

```
Inductive type: Set :=
  | Tparam: name → type
  | Tvar: nat → type
  | Top: type
  | Arrow: type → type → type
  | Forall: type → type → type.
```

The free names of a type are defined as follow. Notice the *Forall* case: *Forall* does not bind any name.

```
Fixpoint fv_type (t: type) : list name :=
  match t with
  | Tparam x ⇒ x :: nil
  | Tvar n ⇒ nil
  | Top ⇒ nil
  | Arrow t1 t2 ⇒ fv_type t1 ++ fv_type t2
  | Forall t1 t2 ⇒ fv_type t1 ++ fv_type t2
  end.
```

There are two substitution operations over types, written *vsubst* and *psubst* in Pollack's talk. *vsubst* substitutes a type for a bound variable (a de Bruijn index). *psubst* substitutes a type for a free variable (a name).

The crucial observation is that variable capture cannot occur during either substitution:

- Types never contain free de Bruijn indices, since these indices are used only for representing bound variables. Therefore, *vsubst* does not need to perform lifting of de Bruijn indices in the substituted type.
- Types never bind names, only de Bruijn indices. Therefore, *psubst* never needs to perform renaming of names in the substituted term when descending below a binder.

```
Fixpoint vsubst_type (a: type) (x: nat) (b: type) {struct a} : type :=
  match a with
  | Tparam n ⇒ Tparam n
  | Tvar n ⇒
      match compare_nat n x with
      | Nat_less _ ⇒ Tvar n
      | Nat_equal _ ⇒ b
      | Nat_greater _ ⇒ Tvar (pred n)
      end
  | Top ⇒ Top
  | Arrow a1 a2 ⇒ Arrow (vsubst_type a1 x b) (vsubst_type a2 x b)
  | Forall a1 a2 ⇒ Forall (vsubst_type a1 x b) (vsubst_type a2 (S x) b)
  end.
```

```
Fixpoint psubst_type (a: type) (x: name) (b: type) {struct a} : type :=
  match a with
  | Tparam n ⇒ if eq_name n x then b else Tparam n
  | Tvar n ⇒ Tvar n
  | Top ⇒ Top
  | Arrow a1 a2 ⇒ Arrow (psubst_type a1 x b) (psubst_type a2 x b)
  | Forall a1 a2 ⇒ Forall (psubst_type a1 x b) (psubst_type a2 x b)
  end.
```

In the remainder of the development, *vsubst* is only used to replace bound variable 0 by a fresh, free variable (a name) when taking apart a *Forall* type. This operation is similar to the "freshening" operation used in Fresh ML and related systems. We call it *freshen_type* for clarity.

```
Definition freshen_type (a: type) (x: name) : type := vsubst_type a 0 (Tparam x).
```

Free variables and freshening play well together.

```
Lemma fv_type_vsubst_type:
  ∀ x a n b, In x (fv_type a) → In x (fv_type (vsubst_type a n b)).
```

```
Lemma fv_type_freshen_type:
  ∀ x a y, In x (fv_type a) → In x (fv_type (freshen_type a y)).
```

We now define swaps (permutation of names) over types and show basic properties of swaps that will be useful later.

```
Fixpoint swap_type (u v: name) (t: type) {struct t} : type :=
  match t with
  | Tparam x ⇒ Tparam (swap u v x)
  | Tvar n ⇒ Tvar n
  | Top ⇒ Top
  | Arrow t1 t2 ⇒ Arrow (swap_type u v t1) (swap_type u v t2)
  | Forall t1 t2 ⇒ Forall (swap_type u v t1) (swap_type u v t2)
  end.
```

Swaps are involutions (self-inverse).

```
Lemma swap_type_inv: ∀ u v t, swap_type u v (swap_type u v t) = t.
```

Swaps of variables that do not occur free in a type leave the type unchanged.

```
Lemma swap_type_not_free:
  ∀ u v t, ¬In u (fv_type t) → ¬In v (fv_type t) → swap_type u v t = t.
```

Swaps commute with *vsubst* substitution and freshening.

Lemma *vsubst_type_swap*:
  ∀ *u v a n b*,
  swap_type u v (*vsubst_type a n b*) = *vsubst_type* (*swap_type u v a*) *n* (*swap_type u v b*).

Lemma *freshen_type_swap*:
  ∀ *u v a x*,
  swap_type u v (*freshen_type a x*) = *freshen_type* (*swap_type u v a*) (*swap u v x*).

Swaps commute with the computation of free variables.

Lemma *in_fv_type_swap*:
  ∀ *u v x t*, *In x* (*fv_type t*) ↔ *In* (*swap u v x*) (*fv_type* (*swap_type u v t*)).

2.2.2 Typing environments

Typing environments are standard: lists of (name, type) pairs. Bindings are added to the left of the environment using the *cons* list operation. Thus, later bindings come first.

Definition *typenv* := *list* (*name* × *type*).

Definition *dom* (*e*: *typenv*) := *map* (@*fst name type*) *e*.

Lemma *dom_append*: ∀ *e1 e2*, *dom* (*e1* ++ *e2*) = *dom e1* ++ *dom e2*.

The *lookup* function returns the type associated with a name in a typing environment.

```
Fixpoint lookup (x: name) (e: typenv) {struct e} : option type :=
  match e with
  | nil ⇒ None
  | (y, t) :: e' ⇒ if eq_name x y then Some t else lookup x e'
  end.
```

Lemma *lookup_inv*: ∀ *x t e*, *lookup x e* = *Some t* → *In x* (*dom e*).

Lemma *lookup_exists*: ∀ *x e*, *In x* (*dom e*) → ∃ *t*, *lookup x e* = *Some t*.

We extend swaps to typing environments, pointwise.

```
Fixpoint swap_env (u v: name) (e: typenv) {struct e} : typenv :=
  match e with
  | nil ⇒ nil
  | (x, t) :: e' ⇒ (swap u v x, swap_type u v t) :: swap_env u v e'
  end.
```

Environment lookup commutes with swaps.

Lemma *lookup_swap*:
  ∀ *u v x e t*, *lookup x e* = *Some t* →
  lookup (*swap u v x*) (*swap_env u v e*) = *Some* (*swap_type u v t*).

The *dom* operation commutes with swaps.

```
Lemma in_dom_swap:
  ∀ u v x e, In x (dom e) ↔ In (swap u v x) (dom (swap_env u v e)).
```

2.2.3 Well-formedness of types and environments

A type is well-formed in a typing environment if:

- all names free in the type are of kind *TYPE*;
- all names free in the type are bound in the environment;
- it does not contain free de Bruijn variables.

We capture these conditions by the following inference rules.

```
Inductive wf_type: typenv → type → Prop :=
  | wf_type_param: ∀ x e,
      kind x = TYPE → In x (dom e) →
      wf_type e (Tparam x)
  | wf_type_top: ∀ e,
      wf_type e Top
  | wf_type_arrow: ∀ e t1 t2,
      wf_type e t1 → wf_type e t2 → wf_type e (Arrow t1 t2)
  | wf_type_forall: ∀ e t1 t2,
      wf_type e t1 →
      (∀ x,
         kind x = TYPE → ¬In x (fv_type t2) → ¬In x (dom e) →
         wf_type ((x, t1) :: e) (freshen_type t2 x)) →
      wf_type e (Forall t1 t2).
```

The rules are straightforward, except perhaps the *wf_type_forall* rule. It follows a general pattern for operating over sub-terms of a binder, such as *t2* in *Forall t1 t2*. The de Bruijn variable *Tvar* 0 is potentially free in *t2*. To recover a well-formed term, without free de Bruijn variables, we substitute *Tvar* 0 with a fresh name *x*. Therefore, the premise for *t2* applies to *freshen_type t2 x*.

How should *x* be chosen? As in the name-based specification, *x* must not be in the domain of *e*, otherwise the extended environment (*x*, *t1*) :: *e* would be ill-formed. In addition, the name *x* must not be free in *t2*, otherwise the freshening *freshen_type t2 x* would incorrectly identify the bound, universally-quantified variable of the *Forall* types with an existing, free type variable.

How should *x* be quantified? That is, should the second premise *wf_type* ((*x*, *t1*) :: *e*) (*freshen_type t2 x*)) hold for one particular name *x* not in *dom*(*e*), or for all names *x* not in *dom*(*e*)? The "for all" alternative obviously leads to a stronger induction principle: proofs that proceed by inversion or induction over an hypothesis *wf_type e* (*Forall t1 t2*) can then choose any convenient x fresh for e to exploit the second premise, rather than having to cope with a fixed, earlier choice of x. Symmetrically, the "for one" alternative is more convenient for proofs that must conclude *wf_type e* (*Forall t1 t2*): it suffices to exhibit one suitable x fresh in e that satisfies the second premise, rather than having to establish the second premise for all such x.

The crucial observation is that those two alternative are equivalent: the same subtyping judgements can be derived with the "for all" rule and the "for one" rule. (See Pollack's talk for more explanations.) Therefore, in the definition of the *wf_type* predicate above, we chose the "for all" rule, so as to get the strongest induction principle. And we will show shortly that the "for one" rule is admissible and can be used in proofs that conclude *wf_type e* (*Forall t1 t2*).

An environment is well-formed if every type it contains is well-formed in the part of the environment that occurs to its right, i.e. the environment at the time this type was introduced. This ensures in particular that all the variables in this type are bound earlier (i.e. to the right) in the environment. Moreover, we impose that no name is bound twice in an environment.

```
Inductive wf_env: typenv → Prop :=
  | wf_env_nil:
      wf_env nil
  | wf_env_cons: ∀ x t e,
      wf_env e → ¬In x (dom e) → wf_type e t →
      wf_env ((x, t) :: e).
```

```
Lemma wf_type_env_incr:
  ∀ e t, wf_type e t → ∀ e', incl (dom e) (dom e') → wf_type e' t.
```

A type well formed in e has all its free names in the domain of e.

```
Lemma fv_wf_type: ∀ x e t, wf_type e t → In x (fv_type t) → In x (dom e).
```

Looking up the type of a name in a well-formed environment returns a well-formed type.

```
Lemma wf_type_lookup: ∀ e, wf_env e → ∀ x t, lookup x e = Some t → wf_type e t.
```

Type well-formedness is stable by swapping.

```
Lemma wf_type_swap:
  ∀ u v e t,
  kind u = kind v → wf_type e t → wf_type (swap_env u v e) (swap_type u v t).
```

Environment well-formedness is stable by swapping.

```
Lemma wf_env_swap:
  ∀ u v e, kind u = kind v → wf_env e → wf_env (swap_env u v e).
```

The domain of an environment is invariant by swaps of names that are not in this domain.

Lemma *swap_env_dom*:
  $\forall$ *u v e*, $\neg$*In u* (*dom e*) $\rightarrow$ $\neg$*In v* (*dom e*) $\rightarrow$ *dom* (*swap_env u v e*) = *dom e*.

A well-formed environment is invariant by swaps of names that are not in the domains of this environment.

Lemma *swap_env_not_free*:
  $\forall$ *u v e*, *wf_env e* $\rightarrow$ $\neg$*In u* (*dom e*) $\rightarrow$ $\neg$*In v* (*dom e*) $\rightarrow$ *swap_env u v e* = *e*.

We now show that the alternate formulation of rule *wf_type_forall* (the one with "for one fresh name x" instead of "for all fresh names x" in the second premise) is admissible.

Lemma *wf_type_forall'*:
  $\forall$ *e x t t1 t2*,
  wf_type e t1 $\rightarrow$ *kind x* = *TYPE* $\rightarrow$ $\neg$*In x* (*fv_type t2*) $\rightarrow$ $\neg$*In x* (*dom e*) $\rightarrow$
  wf_type ((*x*, *t*) :: *e*) (*freshen_type t2 x*) $\rightarrow$
  wf_type e (*Forall t1 t2*).

2.3 Algorithmic subtyping

We now define the subtyping judgement as an inductive predicate. Each constructor of the predicate corresponds to an inference rule in the original definition of subtyping.

Inductive *is_subtype*: *typenv* $\rightarrow$ *type* $\rightarrow$ *type* $\rightarrow$ *Prop* :=
  | *sa_top*: $\forall$ *e s*,
      wf_env e $\rightarrow$ *wf_type e s* $\rightarrow$
      is_subtype e s Top
  | *sa_refl_tvar*: $\forall$ *e x u*,
      wf_env e $\rightarrow$ *kind x* = *TYPE* $\rightarrow$ *lookup x e* = *Some u* $\rightarrow$
      is_subtype e (*Tparam x*) (*Tparam x*)
  | *sa_trans_tvar*: $\forall$ *e x u t*,
      kind x = *TYPE* $\rightarrow$ *lookup x e* = *Some u* $\rightarrow$ *is_subtype e u t* $\rightarrow$
      is_subtype e (*Tparam x*) *t*
  | *sa_arrow*: $\forall$ *e s1 s2 t1 t2*,
      is_subtype e t1 s1 $\rightarrow$ *is_subtype e s2 t2* $\rightarrow$
      is_subtype e (*Arrow s1 s2*) (*Arrow t1 t2*)
  | *sa_all*: $\forall$ *e s1 s2 t1 t2*,
      is_subtype e t1 s1 $\rightarrow$
      ($\forall$ *x*,
        kind x = *TYPE* $\rightarrow$ $\neg$*In x* (*dom e*) $\rightarrow$
        is_subtype ((*x*, *t1*) :: *e*) (*freshen_type s2 x*) (*freshen_type t2 x*)) $\rightarrow$
      is_subtype e (*Forall s1 s2*) (*Forall t1 t2*).

The *sa_all* rule for *Forall* types follows the pattern that we already introduced for the *wf_type* predicate, in rule *wf_type_forall.* In the original, name-based specification, we say that $E \vdash (\forall x <: \sigma_1.\ \sigma_2) <: (\forall x <: \tau_1.\ \tau_2)$ if $E \vdash \tau_1 <: \sigma_1$ and $E,\ x : \tau_1 \vdash \sigma_2 <: \tau_2$. The type variable x, being α-convertible in the conclusion, is (implicitly or explicitly) chosen so that $E,\ x : \tau_1$ is well-formed in the second premise, that is, x is chosen not free in E.

In our locally nameless representation, the type variables bound by *Forall* in the conclusion do not have names. We must therefore invent a suitable name x and substitute it for the bound variable *TVar* 0 in the types *s2* and *t2.* Therefore, the second premise puts *freshen_type s2 x* and *freshen_type t2 x* in subtype relation.

As mentioned already, x should be chosen not in the domain of e (otherwise the extended environment (*x*, *t1*) :: *e* would be ill-formed) and not free in *s2* and *t2*, otherwise the freshenings *freshen_type s2 x* and *freshen_type t2 x* would incorrectly identify the bound, universally-quantified variable of the *Forall* types with an existing, free type variable. However, as we will prove below, the rules for *is_subtype* satisfy a well-formedness condition: if *is_subtype e u1 u2*, then *u1* and *u2* are well-formed in *e*, implying that a name not in the domain of *e* cannot be free in *u1* or *u2.* Therefore, the condition "x not in the domain of e" suffices to ensure that x is not free in *s2* and *t2*, and that the freshenings *freshen_type s2 x* and *freshen_type t2 x* make sense.

As mentioned already as well, we have a choice between quantifying over all suitable x or over one suitable x in the second premise. Again, we go with the "for all" alternative in order to obtain the strongest induction principle, and we will show later that the "for one" alternative is derivable.

For the time being, we start with simple well-formedness properties of the types and environments involved in a *is_subtype* relation.

Lemma *is_subtype_wf_env*: ∀ *e s t*, *is_subtype e s t* → *wf_env e.*

Lemma *is_subtype_wf_type*: ∀ *e s t*, *is_subtype e s t* → *wf_type e s* ∧ *wf_type e t.*

Lemma *is_subtype_wf_type_l*: ∀ *e s t*, *is_subtype e s t* → *wf_type e s.*

Lemma *is_subtype_wf_type_r*: ∀ *e s t*, *is_subtype e s t* → *wf_type e t.*

We now show that the *is_subtype* predicate is stable by swapping. This property is crucial to show the equivalence of the "for all" and "for one" interpretations of rule *sa_all.*

Lemma *is_subtype_swap*:
  ∀ *u v*, *kind u* = *kind v* →
  ∀ *e s t*, *is_subtype e s t* →
  is_subtype (*swap_env u v e*) (*swap_type u v s*) (*swap_type u v t*).

Two silly lemmas about freshness of names in types.

Lemma *fresh_wf_type*: ∀ *x e t*, *wf_type e t* → ¬*In x* (*dom e*) → ¬*In x* (*fv_type t*).

Lemma *fresh_freshen_type*:
  $\forall$ *x t1 e t y*,
  wf_type ((*x*, *t1*) :: *e*) (*freshen_type t x*) $\rightarrow \neg$*In y* (*dom e*) $\rightarrow x \neq y \rightarrow \neg$*In y* (*fv_type t*).

We now show that the alternate presentation of rule *sa_all* (the one with "for one name" in the second premise instead of "for all names") is admissible.

Lemma *sa_all'*:
  $\forall$ *e s1 s2 t1 t2 x*,
  is_subtype e t1 s1 $\rightarrow$
  kind x = *TYPE* $\rightarrow \neg$*In x* (*dom e*) $\rightarrow \neg$*In x* (*fv_type s2*) $\rightarrow \neg$*In x* (*fv_type t2*) $\rightarrow$
  is_subtype ((*x*, *t1*) :: *e*) (*freshen_type s2 x*) (*freshen_type t2 x*) $\rightarrow$
  is_subtype e (*Forall s1 s2*) (*Forall t1 t2*).

2.4 Reflexivity and transitivity of subtyping

We now turn (at last!) to proving the two theorems of part 1 of the POPLmark challenge: reflexivity and transitivity of subtyping.

Reflexivity of subtyping is shown by straightforward induction on the derivation of well-formedness of the type. As noted by McKinna and Pollack, such inductions conveniently replace inductions on the structure of types.

Theorem *sub_refl*: $\forall$ *t e*, *wf_type e t* $\rightarrow$ *wf_env e* $\rightarrow$ *is_subtype e t t*.
Proof.
  induction 1; *intros*.
Case *t* = *Tparam x*
  destruct (*lookup_exists* _ _ *H0*) *as* [t L].
  apply sa_refl_tvar with t; *auto*.
Case *t* = *Top*
  apply sa_top. *auto*. *constructor*.
Case *t* = *Arrow t1 t2*
  apply sa_arrow. *auto*. *auto*.
Case *t* = *Forall t1 t2*
  destruct (*fresh_name TYPE* (*fv_type t2* ++ *dom e*)) *as* [*x* [FRESH KIND]].
  apply sa_all' with x; *eauto*.
  apply H1; *eauto*.
  constructor; *eauto*.
Qed.

We now do some scaffolding work for the proof of transitivity. First, we will need to perform inductions over the size of types. We cannot just do inductions over the structure of types, as the original paper proof did, because in the case of *Forall t1 t2*, we will need to recurse not on *t2* but on *freshen_type t2 x* for some x, which is not a sub-term of *t2*. However, the size of *freshen_type t2 x* is the same as the size of *t2*, so induction over sizes will work.

```
Fixpoint size_type (t: type): nat :=
  match t with
  | Tparam _ ⇒ 0
  | Tvar _ ⇒ 0
  | Top ⇒ 0
  | Arrow t1 t2 ⇒ 1 + size_type t1 + size_type t2
  | Forall t1 t2 ⇒ 1 + size_type t1 + size_type t2
  end.
```

Lemma *vsubst_type_size*: ∀ *x a n*, *size_type* (*vsubst_type a n* (*Tparam x*)) = *size_type a*.

Lemma *freshen_type_size*: ∀ *x a*, *size_type* (*freshen_type a x*) = *size_type a*.

We now define a notion of inclusion between environments that we call "weakening". *e2* weakens *e1* if all the bindings in *e1* are preserved in *e2*; however, *e2* may contain additional, non-interfering bindings.

```
Definition env_weaken (e1 e2: typenv) : Prop :=
  ∀ x t, lookup x e1 = Some t → lookup x e2 = Some t.
```

Lemma *env_weaken_incl_dom*: ∀ *e1 e2*, *env_weaken e1 e2* → *incl* (*dom e1*) (*dom e2*).

The subtyping relation is stable by weakening of the typing environment.

```
Lemma sub_weaken:
  ∀ e s t, is_subtype e s t → ∀ e', wf_env e' → env_weaken e e' → is_subtype e' s t.
```

A special case of weakening is the addition of bindings to an existing typing environment, provided the resulting environment is well-formed.

```
Lemma env_concat_weaken:
  ∀ delta gamma,
  wf_env (delta ++ gamma) → env_weaken gamma (delta ++ gamma).
```

The following lemmas prove useful properties of environments of the form *e1* ++ (*x*, *p*) :: *e2*, that is, all bindings of *e2*, followed by a binding of *p* to *x*, followed by all bindings of *e1*.

Lemma *dom_env_extends*: ∀ *e1 x p q e2*, *dom* (*e1* ++ (*x*, *p*) :: *e2*) = *dom* (*e1* ++ (*x*, *q*) :: *e2*).

```
Lemma wf_env_extends:
  ∀ e2 x p q e1, wf_env (e1 ++ (x, p) :: e2) → wf_type e2 q → wf_env (e1 ++ (x, q) ::
e2).
```

```
Lemma lookup_env_extends:
  ∀ e2 x p q y e1,
  wf_env (e1 ++ (x, q) :: e2) →
  lookup y (e1 ++ (x, p) :: e2) = if eq_name y x then Some p else lookup y (e1 ++ (x,
q) :: e2).
```

Now comes the major result: transitivity and the narrowing property of subtyping, proved simultaneously. The proof follows the structure of the paper proof, with the structural induction on q being replaced by a Peano induction on the size of q.

```
Lemma sub_trans_narrow:
  ∀ n,
     (∀ e s q t,
        size_type q ≤ n →
        is_subtype e s q → is_subtype e q t →
        is_subtype e s t)
  ∧ (∀ x e1 e2 p q r s,
        size_type q ≤ n →
        is_subtype (e1 ++ (x, q) :: e2) r s → is_subtype e2 p q →
        is_subtype (e1 ++ (x, p) :: e2) r s).
Proof.
  intro n0. pattern n0. apply Peano_induction.
  intros size HRsize.
```

Part 1: transitivity

```
  assert (∀ e s q, is_subtype e s q →
            ∀ t, size_type q ≤ size → is_subtype e q t → is_subtype e s t).
```

Sub-induction on the derivation of *is_subtype e s q*

```
  induction 1; intros.
```

Case *sa_top*

```
  inversion H2. apply sa_top. auto. eauto.
```

Case *sa_refl_tvar*

```
  auto.
```

Case *sa_trans_tvar*

```
  apply sa_trans_tvar with u; auto.
```

Case *sa_arrow*

```
  inversion H2.
    apply sa_top. auto. inversion H4. constructor; eauto.
    subst e0; subst s0; subst s3. simpl in H1.
    assert (SZpos: pred size < size). omega.
    elim (HRsize (pred size) SZpos); intros HR1 HR2.
    apply sa_arrow.
```

Application of the outer induction hypothesis to *t1*

```
    apply HR1 with t1; auto. omega.
```

Application of the outer induction hypothesis to *t2*

```
    apply HR1 with t2; auto. omega.
```

Case *sa_forall*

```
  inversion H3.
```

apply sa_top. auto.
apply is_subtype_wf_type_l with (Forall t1 t2). constructor; assumption.
subst e0; subst s0; subst s3. simpl in H2.
assert (SZpos: pred size < size). omega.
elim (HRsize (pred size) SZpos); intros HR1 HR2.

Choice of an appropriately fresh name x

elim (fresh_name TYPE (dom e ++ fv_type t3 ++ fv_type s2)).
intros x [FRESH KIND].
apply sa_all' with x; eauto.

Application of the outer induction hypothesis to *t1*

apply HR1 with t1; auto. omega.

Application of the outer induction hypothesis to *freshen t2 x*

apply HR1 with (freshen_type t2 x).
rewrite freshen_type_size. omega.
change ((x, t0) :: e) with (nil ++ (x, t0) :: e).

Application of the narrowing part of the outer induction hypothesis.

apply HR2 with t1. omega.
simpl. apply H0; eauto. auto. apply H9; eauto.

Part 2: narrowing

assert (∀ *e r s*,
is_subtype e r s →
∀ *e1 x q e2 p*,
e = *e1* ++ (*x*, *q*) :: *e2* → *size_type q* ≤ *size* → *is_subtype e2 p q* →
is_subtype (*e1* ++ (*x*, *p*) :: *e2*) *r s*).

Sub-induction on the derivation of *is_subtype e r s*

induction 1; *intros*; *subst e.*

Case *sa_top*

apply sa_top. apply wf_env_extends with q; eauto.
apply wf_type_env_incr with (e1 ++ (x, q) :: e2); auto.
rewrite (dom_env_extends e1 x q p e2). apply incl_refl.

Case *sa_refl_tvar*

apply sa_refl_tvar with (if eq_name x x0 then p else u).
apply wf_env_extends with q; eauto. auto.
rewrite (lookup_env_extends e2 x0 p q x e1 H0).
case (eq_name x x0); auto.

Case sa_trans_tvar

apply sa_trans_tvar with (if eq_name x x0 then p else u).
auto. rewrite (lookup_env_extends e2 x0 p q x e1).
case (eq_name x x0); auto. eauto.
case (eq_name x x0); intro.

sub-case *x* = *x0*

generalize H1. rewrite (lookup_env_extends e2 x0 q q x e1); eauto.

```
  rewrite e; rewrite eq_name_true; intro EQ; injection EQ; intro; subst u.
  apply H with q.
  apply sub_weaken with e2; auto.
  apply wf_env_extends with q; eauto.
  change (e1 ++ (x0, p) :: e2) with (e1 ++ (((x0, p) :: nil) ++ e2)).
  rewrite ← app_ass. apply env_concat_weaken.
  rewrite app_ass. simpl. apply wf_env_extends with q; eauto.
  auto.
  apply IHis_subtype with q; auto.
sub-case x ≠ x0
  apply IHis_subtype with q; auto.
Case sa_arrow
  apply sa_arrow.
  apply IHis_subtype1 with q; auto.
  apply IHis_subtype2 with q; auto.
Case sa_forall
  apply sa_all.
  apply IHis_subtype with q; auto.
  intros.
  change ((x0, t1) :: e1 ++ (x, p) :: e2)
    with (((x0, t1) :: e1) ++ (x, p) :: e2).
  apply H2 with q. auto.
  rewrite (dom_env_extends e1 x q p e2). auto.
  reflexivity. auto. auto.
Combining the two parts together
  split. intros; apply H with q; auto.
  intros; apply H0 with (e1 ++ (x, q) :: e2) q; auto.
Qed.
```

As a corollary, we obtain transitivity of subtyping.

```
Theorem sub_trans:
  ∀ e s q t, is_subtype e s q → is_subtype e q t → is_subtype e s t.
```

As well as narrowing.

```
Theorem sub_narrow:
  ∀ x e1 e2 p q r s,
  is_subtype (e1 ++ (x, q) :: e2) r s → is_subtype e2 p q →
  is_subtype (e1 ++ (x, p) :: e2) r s.
```

2.5 Stability of the subtyping judgement under substitutions

We now prove a property of the subtyping judgement that plays a crucial role later to prove that instantiation of polymorphic types is semantically sound. This property is stability under substitutions, namely that if $\Gamma, X <: U \vdash T_1 <: T_2$, then it must be the case that $\Gamma \vdash T_1[X \leftarrow U] <: T_2[X \leftarrow U]$. In the POPLmark challenge, this property belongs to part 2 of the challenge, but we prefer to prove it now, since it involves only the subtyping judgement.

To prove stability under substitution by induction, we need to strengthen its statement as follows: if $\Gamma, X <: U, \Gamma' \vdash T_1 <: T_2$, then $\Gamma, \Gamma'[X \leftarrow U] \vdash T_1[X \leftarrow U] <: T_2[X \leftarrow U]$. We therefore need to extend type substitutions to environments, pointwise.

```
Fixpoint psubst_env (e: typenv) (x: name) (b: type) {struct e} : typenv :=
  match e with
  | nil ⇒ nil
  | (y, t) :: e' ⇒ (y, psubst_type t x b) :: psubst_env e' x b
  end.
```

```
Lemma dom_psubst_env: ∀ x b e, dom (psubst_env e x b) = dom e.
```

2.5.1 Commutation properties for type substitutions

As a preliminary result, we need to show that well-formedness of types and environments is preserved by substitution of a well-formed type for a name. This simple property needs a number of commutation properties between the *psubst_type*, *freshen_type* and *vsubst_type* functions. Unfortunately, some hacking on de Bruijn indices is necessary. We start by defining a predicate *type_vars_below t n* that holds if all free de Bruijn variables in the type t are less than n. In particular, *type_vars_below t* 0 holds iff t has no free de Bruijn variables.

```
Fixpoint type_vars_below (t: type) (n: nat) {struct t} : Prop :=
  match t with
  | Tparam x ⇒ True
  | Tvar m ⇒ m < n
  | Top ⇒ True
  | Arrow t1 t2 ⇒ type_vars_below t1 n ∧ type_vars_below t2 n
  | Forall t1 t2 ⇒ type_vars_below t1 n ∧ type_vars_below t2 (S n)
  end.
```

```
Lemma type_vars_below_vsubst:
  ∀ t n t', type_vars_below (vsubst_type t n t') n → type_vars_below t (S n).
```

A well-formed type, having no free de Bruijn variables, has all its de Bruijn variables below 0.

Lemma *wf_type_vars_below_0*: $\forall$ *e t*, *wf_type e t* $\rightarrow$ *type_vars_below t* 0.

A type is invariant by *vsubst_type* substitution if the de Bruijn variable being substituted is not free in the type.

Lemma *vsubst_invariant_below*:
  $\forall$ *t n m s*, *type_vars_below t n* $\rightarrow$ $n \leq m$ $\rightarrow$ *vsubst_type t m s* = *t*.

As a corollary, well-formed types are invariant by *vsubst* substitutions.

Lemma *vsubst_wf_type*: $\forall$ *e t n s*, *wf_type e t* $\rightarrow$ *vsubst_type t n s* = *t*.

It follows that *vsubst* and *psubst* substitutions commute in the following sense.

Lemma *psubst_vsubst_type*:
  $\forall$ *e a x b n c*,
  wf_type e b $\rightarrow$
  vsubst_type (*psubst_type a x b*) *n* (*psubst_type c x b*) =
  psubst_type (*vsubst_type a n c*) *x b*.

Consequently, *psubst_type* and *freshen_type* commute if they operate on distinct names.

Lemma *psubst_freshen_type*:
  $\forall$ *e a x b y*,
  wf_type e b $\rightarrow$ $x \neq y$ $\rightarrow$
  freshen_type (*psubst_type a x b*) *y* = *psubst_type* (*freshen_type a y*) *x b*.

Additionally, *vsubst* substitution of de Bruijn variable 0 is equivalent to freshening with a fresh type name *x* followed by a *psubst* substitution over *x*.

Lemma *vsubst_psubst_type*:
  $\forall$ *x t2 t1 n*, $\neg$*In x* (*fv_type t1*) $\rightarrow$
  vsubst_type t1 n t2 = *psubst_type* (*vsubst_type t1 n* (*Tparam x*)) *x t2*.

Lemma *vsubst_psubst_freshen_type*:
  $\forall$ *x t1 t2*, $\neg$*In x* (*fv_type t1*) $\rightarrow$
  vsubst_type t1 0 *t2* = *psubst_type* (*freshen_type t1 x*) *x t2*.

2.5.2 Preservation of well-formedness of types and environments during substitution

Well-formedness of types is preserved by *psubst_type* substitution of a well-formed type for a type name.

Lemma *wf_type_psubst*:

  ∀ *e1 x p q e2 t*,
  wf_type (*e2* ++ (*x*, *p*) :: *e1*) *t* → *wf_type e1 q* →
  wf_type (*psubst_env e2 x q* ++ *e1*) (*psubst_type t x q*).

Similarly, well-formedness of environments is preserved by *psubst_env* substitution.

Lemma *wf_env_psubst*:
  ∀ *e1 x p q e2*,
  wf_env (*e2* ++ (*x*, *p*) :: *e1*) → *wf_type e1 q* →
  wf_env (*psubst_env e2 x q* ++ *e1*).

2.5.3 Type substitution preserves subtyping

We now show that if *s* is subtype of *t* in the environment *e2* ++ (*x*, *p*) :: *e1*, and if *q* is subtype of *p* in *e1*, then *psubst s x q* is subtype of *psubst t x q* in *psubst_env x q e2* ++ *e1*.

Well-formed environments of the form *e2* ++ (*x*, *p*) :: *e1* are such that *x* is not in the domain of *e1* and not free in the types listed in *e1*. Therefore, these types are invariant by substitution over *x*.

Lemma *env_concat_not_free*: ∀ *e1 x p e2*, *wf_env* (*e2* ++ (*x*, *p*) :: *e1*) → ¬*In x* (*dom e1*).

Lemma *psubst_type_inv*: ∀ *t x s*, ¬*In x* (*fv_type t*) → *psubst_type t x s* = *t*.

The following technical lemma relates *lookup* operations between the original environment *e2* ++ (*x*, *p*) :: *e1* and its substituted counterpart, *psubst_env e2 x q* ++ *e1*.

Lemma *lookup_env_concat*:
  ∀ *e1 x p q y t e2*,
  wf_env (*e2* ++ (*x*, *p*) :: *e1*) →
  lookup y (*e2* ++ (*x*, *p*) :: *e1*) = *Some t* →
  (*x* = *y* ∧ *t* = *p*) ∨
  (*x* ≠ *y* ∧ *lookup y* (*psubst_env e2 x q* ++ *e1*) = *Some* (*psubst_type t x q*)).

We can now prove the stability of subtyping by type substitution (lemma A.10 in the challenge statement). The proof proceeds by induction over the derivation of the second subtyping hypothesis.

Theorem *sub_stable_subst*:
  ∀ *e1 x p q e2 s t*,
  is_subtype e1 p q →
  is_subtype (*e2* ++ (*x*, *q*) :: *e1*) *s t* →
  is_subtype (*psubst_env e2 x p* ++ *e1*) (*psubst_type s x p*) (*psubst_type t x p*).
Proof.
  assert (∀ *e1 x p q*, *is_subtype e1 p q* →
            ∀ *e s t*, *is_subtype e s t* →

```
           ∀ e2, e = e2 ++ (x, q) :: e1 →
           is_subtype (psubst_env e2 x p ++ e1)
                       (psubst_type s x p) (psubst_type t x p)).
  induction 2; intros; simpl; subst e.
Rule sa_top
  constructor. eapply wf_env_psubst; eauto.
  eapply wf_type_psubst; eauto.
Rule sa_refl
  apply sub_refl.
  case (eq_name x0 x); intro.
Rule sa_refl, case x = x0
  apply wf_type_env_incr with e1; eauto.
  rewrite dom_append. apply incl_appr. apply incl_refl.
Rule sa_refl, case x ≠ x0
  constructor. auto. generalize (lookup_inv _ _ _ H2).
  repeat rewrite dom_append. simpl. rewrite dom_psubst_env.
  intro. apply in_or_app. elim (in_app_or _ _ _ H3); intro.
  auto. elim H4; intro. congruence. auto.
  eapply wf_env_psubst; eauto.
Rule sa_trans
  assert (wf_env (e2 ++ (x, q) :: e1)). eapply is_subtype_wf_env; eauto.
  generalize (lookup_env_concat _ _ _ p _ _ _ H3 H1). intro.
  case (eq_name x0 x); intro.
Rule sa_trans, case x = x0
  subst x0. assert (u = q). intuition. subst u.
  apply sub_trans with (psubst_type q x p).
  replace (psubst_type q x p) with q.
  apply sub_weaken with e1. auto. eapply wf_env_psubst; eauto.
  apply env_concat_weaken. eapply wf_env_psubst; eauto.
  symmetry. apply psubst_type_inv.
  assert (wf_type e1 q). eapply is_subtype_wf_type_r; eauto.
  generalize (fv_wf_type x _ _ H5).
  generalize (env_concat_not_free _ _ _ _ H3). tauto.
  apply IHis_subtype. auto.
Rule sa_trans, case x ≠ x0
  apply sa_trans_tvar with (psubst_type u x p).
  auto. generalize (sym_not_equal n); tauto.
  apply IHis_subtype. auto.
Rule sa_arrow
  constructor; auto.
Rule sa_forall
  destruct (fresh_name TYPE (x :: dom (e2 ++ (x, q) :: e1) ++
                                        dom (psubst_env e2 x p ++ e1) ++
                                        fv_type (psubst_type s2 x p) ++
                                        fv_type (psubst_type t2 x p)))
  as [y [F K]].
  eapply sa_all'; eauto.
  repeat rewrite (psubst_freshen_type e1).
  change ((y, psubst_type t1 x p) :: psubst_env e2 x p ++ e1)
    with (psubst_env ((y, t1) :: e2) x p ++ e1).
  apply H2; eauto. eauto. eauto. eauto. eauto.

  eauto.
Qed.
```

Chapter 3

Type soundness

This chapter addresses part 2A of the POPLmark challenge, namely the proof of soundness of the $F_{<:}$ type systems without records.

```
Require Import Arith.
Require Import ZArith.
Require Import List.
Require Import extralibrary.
Require Import part1a.
```

3.1 Terms

We now define the syntax of $F_{<:}$ terms and basic syntactic notions such as free variables, substitutions, and well-formedness of terms. We follow the same approach used for types in chapter 2.

3.1.1 Syntax and syntactic operations

The syntax of terms is defined as follows. As in types, bound variables are represented by de Bruijn indices, while free variables are represented by names. Bound term variables and bound type variables are numbered independently. In a lambda-abstraction *TFun t a*, the term variable *Var* 0 is bound in *a*. In a type abstraction *TApp t a*, the type variable *TVar* 0 is bound in *a*.

```
Inductive term: Set :=
  | Param: name → term
  | Var: nat → term
  | Fun: type → term → term
  | App: term → term → term
  | TFun: type → term → term
  | TApp: term → type → term.
```

The free names of a term include both type names and term names.

```
Fixpoint fv_term (a: term) : list name :=
  match a with
  | Param v ⇒ v :: nil
  | Var n ⇒ nil
  | Fun t a1 ⇒ fv_type t ++ fv_term a1
  | App a1 a2 ⇒ fv_term a1 ++ fv_term a2
  | TFun t a1 ⇒ fv_type t ++ fv_term a1
  | TApp a1 t ⇒ fv_term a1 ++ fv_type t
  end.
```

There are 4 substitution operations over terms, depending on whether we are substituting a named variable (*psubst_*) or a de Bruijn variable (*vsubst_*), and whether we are substituting a term for a term variable (*_term*) or a type for a type variable (*_tety*).

```
Fixpoint vsubst_term (a: term) (x: nat) (b: term) {struct a} : term :=
  match a with
  | Param v ⇒ Param v
  | Var n ⇒
      match compare_nat n x with
      | Nat_less _ ⇒ Var n
      | Nat_equal _ ⇒ b
      | Nat_greater _ ⇒ Var (pred n)
      end
  | Fun t a1 ⇒ Fun t (vsubst_term a1 (S x) b)
  | App a1 a2 ⇒ App (vsubst_term a1 x b) (vsubst_term a2 x b)
  | TFun t a1 ⇒ TFun t (vsubst_term a1 x b)
  | TApp a1 t ⇒ TApp (vsubst_term a1 x b) t
  end.
```

```
Fixpoint psubst_term (a: term) (x: name) (b: term) {struct a} : term :=
  match a with
  | Param v ⇒ if eq_name v x then b else Param v
  | Var n ⇒ Var n
  | Fun t a1 ⇒ Fun t (psubst_term a1 x b)
  | App a1 a2 ⇒ App (psubst_term a1 x b) (psubst_term a2 x b)
  | TFun t a1 ⇒ TFun t (psubst_term a1 x b)
  | TApp a1 t ⇒ TApp (psubst_term a1 x b) t
  end.
```

```
Fixpoint vsubst_tety (a: term) (x: nat) (b: type) {struct a} : term :=
  match a with
  | Param v ⇒ Param v
  | Var n ⇒ Var n
  | Fun t a1 ⇒ Fun (vsubst_type t x b) (vsubst_tety a1 x b)
  | App a1 a2 ⇒ App (vsubst_tety a1 x b) (vsubst_tety a2 x b)
  | TFun t a1 ⇒ TFun (vsubst_type t x b) (vsubst_tety a1 (S x) b)
  | TApp a1 t ⇒ TApp (vsubst_tety a1 x b) (vsubst_type t x b)
  end.
```

```
Fixpoint psubst_tety (a: term) (x: name) (b: type) {struct a} : term :=
  match a with
  | Param v ⇒ Param v
  | Var n ⇒ Var n
  | Fun t a1 ⇒ Fun (psubst_type t x b) (psubst_tety a1 x b)
  | App a1 a2 ⇒ App (psubst_tety a1 x b) (psubst_tety a2 x b)
  | TFun t a1 ⇒ TFun (psubst_type t x b) (psubst_tety a1 x b)
  | TApp a1 t ⇒ TApp (psubst_tety a1 x b) (psubst_type t x b)
  end.
```

Here are the two "freshening" operations that replace the bound variable 0 with a term or type name, respectively.

```
Definition freshen_term (a: term) (x: name) : term :=
  vsubst_term a 0 (Param x).
```

```
Definition freshen_tety (a: term) (x: name) : term :=
  vsubst_tety a 0 (Tparam x).
```

Substitutions and freshening play well with free variables.

```
Lemma fv_term_vsubst_term:
  ∀ x a n b, In x (fv_term a) → In x (fv_term (vsubst_term a n b)).
```

```
Lemma fv_term_vsubst_tety:
  ∀ x a n b, In x (fv_term a) → In x (fv_term (vsubst_tety a n b)).
```

```
Lemma fv_term_freshen_term:
  ∀ x a y, In x (fv_term a) → In x (fv_term (freshen_term a y)).
```

```
Lemma fv_term_freshen_tety:
  ∀ x a y, In x (fv_term a) → In x (fv_term (freshen_tety a y)).
```

Swaps of two names in a term.

```
Fixpoint swap_term (u v: name) (a: term) {struct a} : term :=
  match a with
  | Param x ⇒ Param (swap u v x)
  | Var n ⇒ Var n
```

$\mid$ *Fun t a1* $\Rightarrow$ *Fun* (*swap_type u v t*) (*swap_term u v a1*)
$\mid$ *App a1 a2* $\Rightarrow$ *App* (*swap_term u v a1*) (*swap_term u v a2*)
$\mid$ *TFun t a1* $\Rightarrow$ *TFun* (*swap_type u v t*) (*swap_term u v a1*)
$\mid$ *TApp a1 t* $\Rightarrow$ *TApp* (*swap_term u v a1*) (*swap_type u v t*)
end.

Swaps commute with the free variables operation.

Lemma *in_fv_term_swap*:
$\forall$ *u v x a*,
In x (*fv_term a*) $\leftrightarrow$ *In* (*swap u v x*) (*fv_term* (*swap_term u v a*)).

Lemma *swap_term_not_free*:
$\forall$ *u v a*, $\neg$*In u* (*fv_term a*) $\rightarrow$ $\neg$*In v* (*fv_term a*) $\rightarrow$ *swap_term u v a* = *a*.

Swaps are self-inverse.

Lemma *swap_term_inv*: $\forall$ *u v a*, *swap_term u v* (*swap_term u v a*) = *a*.

Swaps commute with substitutions and freshening.

Lemma *vsubst_term_swap*:
$\forall$ *u v a n b*,
swap_term u v (*vsubst_term a n b*) =
vsubst_term (*swap_term u v a*) *n* (*swap_term u v b*).

Lemma *vsubst_tety_swap*:
$\forall$ *u v a n b*,
swap_term u v (*vsubst_tety a n b*) =
vsubst_tety (*swap_term u v a*) *n* (*swap_type u v b*).

Lemma *freshen_term_swap*:
$\forall$ *u v a x*,
swap_term u v (*freshen_term a x*) =
freshen_term (*swap_term u v a*) (*swap u v x*).

Lemma *freshen_tety_swap*:
$\forall$ *u v a x*,
swap_term u v (*freshen_tety a x*) =
freshen_tety (*swap_term u v a*) (*swap u v x*).

3.1.2 Well-formedness of terms

A term is well-formed in a typing environment if:

- all types contained within are well-formed as per *wf_type*;

- all names n appearing free in a *Param n* subterm are of kind *TERM* and are bound in the environment;
- it does not contain free de Bruijn variables.

```
Inductive wf_term: typenv → term → Prop :=
  | wf_term_param: ∀ e x,
      kind x = TERM → In x (dom e) →
      wf_term e (Param x)
  | wf_term_fun: ∀ e t a,
      wf_type e t →
      (∀ x,
       kind x = TERM →
       ¬In x (fv_term a) → ¬In x (dom e) →
       wf_term ((x, t) :: e) (freshen_term a x)) →
      wf_term e (Fun t a)
  | wf_term_app: ∀ e a1 a2,
      wf_term e a1 → wf_term e a2 →
      wf_term e (App a1 a2)
  | wf_term_tfun: ∀ e t a,
      wf_type e t →
      (∀ x,
       kind x = TYPE →
       ¬In x (fv_term a) → ¬In x (dom e) →
       wf_term ((x, t) :: e) (freshen_tety a x)) →
      wf_term e (TFun t a)
  | wf_term_tapp: ∀ e a t,
      wf_term e a → wf_type e t →
      wf_term e (TApp a t).
```

A term well formed in e has all its free names in the domain of e.

```
Lemma fv_wf_term: ∀ x e t, wf_term e t → In x (fv_term t) → In x (dom e).
```

Well-formedness is stable under swaps.

```
Lemma wf_term_swap:
  ∀ u v, kind u = kind v →
  ∀ e a, wf_term e a → wf_term (swap_env u v e) (swap_term u v a).
```

A term well-formed in e remains well-formed if extra bindings are added to e.

```
Lemma wf_term_env_incr:
  ∀ e a, wf_term e a → ∀ e', incl (dom e) (dom e') → wf_term e' a.
```

Here are two admissible rules that prove the well-formedness of *Fun* and *TFun* abstractions. These rules are similar to the *wf_term_fun* and *wf_term_tfun* rules, but with a premise of the form “there exists a name” instead of the original “for all names”.

```
Lemma wf_term_fun':
  ∀ e x t a,
  wf_type e t →
  kind x = TERM → ¬In x (fv_term a) → ¬In x (dom e) →
  wf_term ((x, t) :: e) (freshen_term a x) →
  wf_term e (Fun t a).

Lemma wf_term_tfun':
  ∀ e x t a,
  wf_type e t →
  kind x = TYPE → ¬In x (fv_term a) → ¬In x (dom e) →
  wf_term ((x, t) :: e) (freshen_tety a x) →
  wf_term e (TFun t a).
```

3.1.3 Properties of term substitutions

To prove the usual properties of term substitutions, we follow the same approach as for type substitutions, starting with a characterization of terms that have no free de Bruijn variables, or all such variables below some threshold.

```
Fixpoint term_vars_below (a: term) (nterm ntype: nat) {struct a} : Prop :=
  match a with
  | Param x ⇒ True
  | Var n ⇒ n < nterm
  | Fun t b ⇒ type_vars_below t ntype ∧ term_vars_below b (S nterm) ntype
  | App b c ⇒ term_vars_below b nterm ntype ∧ term_vars_below c nterm ntype
  | TFun t b ⇒ type_vars_below t ntype ∧ term_vars_below b nterm (S ntype)
  | TApp b t ⇒ term_vars_below b nterm ntype ∧ type_vars_below t ntype
  end.

Lemma term_vars_below_vsubst_term:
  ∀ a nterm ntype a',
  term_vars_below (vsubst_term a nterm a') nterm ntype →
  term_vars_below a (S nterm) ntype.

Lemma term_vars_below_vsubst_tety:
  ∀ a nterm ntype a',
  term_vars_below (vsubst_tety a ntype a') nterm ntype →
  term_vars_below a nterm (S ntype).

Lemma wf_term_vars_below_0: ∀ e a, wf_term e a → term_vars_below a 0 0.

Lemma vsubst_term_invariant_below:
  ∀ a n1 n2 m b, term_vars_below a n1 n2 → n1 ≤ m → vsubst_term a m b = a.
```

Lemma *vsubst_tety_invariant_below*:
  $\forall$ *a n1 n2 m t*, *term_vars_below a n1 n2* $\rightarrow$ *n2* $\leq$ *m* $\rightarrow$ *vsubst_tety a m t* = *a*.

Lemma *vsubst_term_wf_term*:
  $\forall$ *e a n b*, *wf_term e a* $\rightarrow$ *vsubst_term a n b* = *a*.

Lemma *vsubst_tety_wf_term*:
  $\forall$ *e a n t*, *wf_term e a* $\rightarrow$ *vsubst_tety a n t* = *a*.

Lemma *psubst_vsubst_term*:
  $\forall$ *e a x b n c*,
  wf_term e b $\rightarrow$
  vsubst_term (*psubst_term a x b*) *n* (*psubst_term c x b*) =
  psubst_term (*vsubst_term a n c*) *x b*.

Lemma *psubst_freshen_term*:
  $\forall$ *e a x b y*,
  wf_term e b $\rightarrow$ *x* $\neq$ *y* $\rightarrow$
  freshen_term (*psubst_term a x b*) *y* = *psubst_term* (*freshen_term a y*) *x b*.

Lemma *psubst_vsubst_tety*:
  $\forall$ *e a x b n c*,
  wf_type e b $\rightarrow$
  vsubst_tety (*psubst_tety a x b*) *n* (*psubst_type c x b*) =
  psubst_tety (*vsubst_tety a n c*) *x b*.

Lemma *psubst_freshen_tety*:
  $\forall$ *e a x b y*,
  wf_type e b $\rightarrow$ *x* $\neq$ *y* $\rightarrow$
  freshen_tety (*psubst_tety a x b*) *y* = *psubst_tety* (*freshen_tety a y*) *x b*.

Lemma *psubst_vsubst_tetety*:
  $\forall$ *e a x b n c*,
  wf_type e b $\rightarrow$
  vsubst_term (*psubst_tety a x b*) *n* (*psubst_tety c x b*) =
  psubst_tety (*vsubst_term a n c*) *x b*.

Lemma *psubst_freshen_tetety*:
  $\forall$ *e a x b y*,
  wf_type e b $\rightarrow$ *x* $\neq$ *y* $\rightarrow$
  freshen_term (*psubst_tety a x b*) *y* = *psubst_tety* (*freshen_term a y*) *x b*.

Lemma *psubst_vsubst_tetyte*:
  $\forall$ *e a x b n c*,
  wf_term e b $\rightarrow$
  vsubst_tety (*psubst_term a x b*) *n c* = *psubst_term* (*vsubst_tety a n c*) *x b*.

Lemma *psubst_freshen_tetyte*:

```
  ∀ e a x b y,
  wf_term e b → x ≠ y →
  freshen_tety (psubst_term a x b) y = psubst_term (freshen_tety a y) x b.
Lemma vsubst_psubst_term:
  ∀ x a2 a1 n,
  ¬In x (fv_term a1) →
  vsubst_term a1 n a2 = psubst_term (vsubst_term a1 n (Param x)) x a2.
Lemma vsubst_psubst_freshen_term:
  ∀ x a1 a2,
  ¬In x (fv_term a1) →
  vsubst_term a1 0 a2 = psubst_term (freshen_term a1 x) x a2.
Lemma vsubst_psubst_tety:
  ∀ x t2 a1 n,
  ¬In x (fv_term a1) →
  vsubst_tety a1 n t2 = psubst_tety (vsubst_tety a1 n (Tparam x)) x t2.
Lemma vsubst_psubst_freshen_tety:
  ∀ x a1 t2,
  ¬In x (fv_term a1) →
  vsubst_tety a1 0 t2 = psubst_tety (freshen_tety a1 x) x t2.
```

3.2 Typing rules

We now define the typing judgement "term a has type t in environment e" as an inductive predicate *has_type e a t*.

```
Inductive has_type: typenv → term → type → Prop :=
  | t_var: ∀ e x t,
      wf_env e → kind x = TERM → lookup x e = Some t →
      has_type e (Param x) t
  | t_abs: ∀ e t1 a t2,
      wf_type e t1 →
      (∀ x,
        kind x = TERM → ¬In x (dom e) →
        has_type ((x, t1) :: e) (freshen_term a x) t2) →
      has_type e (Fun t1 a) (Arrow t1 t2)
  | t_app: ∀ e a b t1 t2,
      has_type e a (Arrow t1 t2) → has_type e b t1 →
      has_type e (App a b) t2
  | t_tabs: ∀ e t1 a t2,
      wf_type e t1 →
      (∀ x,
        kind x = TYPE → ¬In x (dom e) →
        has_type ((x, t1) :: e) (freshen_tety a x) (freshen_type t2 x)) →
      has_type e (TFun t1 a) (Forall t1 t2)
  | t_tapp: ∀ e a t t1 t2,
      has_type e a (Forall t1 t2) →
      is_subtype e t t1 →
      has_type e (TApp a t) (vsubst_type t2 O t)
  | t_sub: ∀ e a t1 t2,
      has_type e a t1 → is_subtype e t1 t2 →
      has_type e a t2.
```

Well-formedness properties: if *has_type e a t* holds, then *e* is a well-formed environment, *t* a well-formed type and *a* a well-formed term.

Lemma *has_type_wf_env*: ∀ *e a t*, *has_type e a t* → *wf_env e*.

Lemma *wf_type_strengthen*:
  ∀ *e t*, *wf_type e t* →
  ∀ *e'*, (∀ *x*, *kind x* = *TYPE* → *In x* (*dom e*) → *In x* (*dom e'*)) → *wf_type e' t*.

Lemma *has_type_wf_type*: ∀ *e a t*, *has_type e a t* → *wf_type e t*.

Lemma *has_type_wf_term*: ∀ *e a t*, *has_type e a t* → *wf_term e a*.

The *has_type* predicate is stable by addition of hypotheses.

Lemma *wf_type_weaken*: ∀ *e e' t*, *wf_type e t* → *env_weaken e e'* → *wf_type e' t*.

Lemma *wf_term_weaken*: ∀ *e e' a*, *wf_term e a* → *env_weaken e e'* → *wf_term e' a*.

Lemma *env_weaken_add*:
  ∀ *e e' x t*, *env_weaken e e'* → *env_weaken* ((*x*, *t*) :: *e*) ((*x*, *t*) :: *e'*).

Lemma *has_type_weaken*:
  ∀ *e a t*, *has_type e a t* → ∀ *e'*, *wf_env e'* → *env_weaken e e'* → *has_type e' a t*.

The *has_type* predicate is equivariant, i.e. stable by swapping.

Lemma *has_type_swap*:
  ∀ *u v*, *kind u* = *kind v* →
  ∀ *e a t*, *has_type e a t* →
  has_type (*swap_env u v e*) (*swap_term u v a*) (*swap_type u v t*).

As a consequence of equivariance, we obtain admissible typing rules for functions and type abstractions, similar to rules *t_abs* and *t_tabs* but where the variable name is quantified existentially rather than universally.

Lemma *kind_fv_type*: ∀ *e t*, *wf_type e t* → ∀ *x*, *In x* (*fv_type t*) → *kind x* = *TYPE*.

Lemma *fv_wf_type_kind*: $\forall$ *x e t*, *wf_type e t* $\rightarrow$ *kind x* = *TERM* $\rightarrow$ $\neg$*In x* (*fv_type t*).

Lemma *fresh_freshen_term*:
  $\forall$ *x t1 e a y*,
  wf_term ((*x*, *t1*) :: *e*) (*freshen_term a x*) $\rightarrow$ $\neg$*In y* (*dom e*) $\rightarrow$ $x \neq y$ $\rightarrow$
  $\neg$*In y* (*fv_term a*).

Lemma *t_abs'*:
  $\forall$ *e t1 a t2 x*,
  kind x = *TERM* $\rightarrow$ $\neg$*In x* (*dom e*) $\rightarrow$ $\neg$*In x* (*fv_term a*) $\rightarrow$
  has_type ((*x*, *t1*) :: *e*) (*freshen_term a x*) *t2* $\rightarrow$
  has_type e (*Fun t1 a*) (*Arrow t1 t2*).

Lemma *fresh_freshen_tety*:
  $\forall$ *x t1 e a y*,
  wf_term ((*x*, *t1*) :: *e*) (*freshen_tety a x*) $\rightarrow$ $\neg$*In y* (*dom e*) $\rightarrow$ $x \neq y$ $\rightarrow$
  $\neg$*In y* (*fv_term a*).

Lemma *t_tabs'*:
  $\forall$ *e t1 a t2 x*,
  kind x = *TYPE* $\rightarrow$ $\neg$*In x* (*dom e*) $\rightarrow$ $\neg$*In x* (*fv_term a*) $\rightarrow$ $\neg$*In x* (*fv_type t2*) $\rightarrow$
  has_type ((*x*, *t1*) :: *e*) (*freshen_tety a x*) (*freshen_type t2 x*) $\rightarrow$
  has_type e (*TFun t1 a*) (*Forall t1 t2*).

3.3 Stability of the typing judgement under substitutions

We now show that the typing judgement is stable under substitutions. There are two substitutions to consider: of a type for a type variable, and of a term for a term variable.

Lemma *has_type_stable_type_subst*:
  $\forall$ *e1 x p q e2 a t*,
  kind x = *TYPE* $\rightarrow$
  is_subtype e1 p q $\rightarrow$
  has_type (*e2* ++ (*x*, *q*) :: *e1*) *a t* $\rightarrow$
  has_type (*psubst_env e2 x p* ++ *e1*) (*psubst_tety a x p*) (*psubst_type t x p*).

Lemma *lookup_env_append*:
  $\forall$ *e2 x p y e1*,
  wf_env (*e1* ++ (*x*, *p*) :: *e2*) $\rightarrow$
  lookup y (*e1* ++ (*x*, *p*) :: *e2*) = *if eq_name y x then Some p else lookup y* (*e1* ++ *e2*).

Lemma *wf_env_append*:
  $\forall$ *e2 x p e1*, *wf_env* (*e1* ++ (*x*, *p*) :: *e2*) $\rightarrow$ *kind x* = *TERM* $\rightarrow$ *wf_env* (*e1* ++ *e2*).

Lemma *is_subtype_strengthen*:

```
  ∀ e s t, is_subtype e s t →
  ∀ e', wf_env e' → (∀ x : name, kind x = TYPE → lookup x e' = lookup x e) →
  is_subtype e' s t.
```

```
Lemma has_type_stable_term_subst:
  ∀ e1 x b s e2 a t,
  kind x = TERM → has_type e1 b s → has_type (e2 ++ (x, s) :: e1) a t →
  has_type (e2 ++ e1) (psubst_term a x b) t.
```

3.4 Dynamic semantics

The dynamic semantics of $F_{<:}$ is specified by a one-step reduction relation, in small-step operational style. We first define values (final results of reduction sequences) as a subset of terms.

```
Inductive isvalue: term → Prop :=
  | isvalue_fun: ∀ t a,
      isvalue (Fun t a)
  | isvalue_tfun: ∀ t a,
      isvalue (TFun t a).
```

We first give a Plotkin-style specification of the reduction relation: it uses inductive rules *red_appfun*, *red_apparg*, *red_tapp* instead of contexts to describe reductions inside applications. The two rules *red_appabs* and *red_tapptabs* are the familiar beta-reduction rules for term and type applications, respectively.

```
Inductive red: term → term → Prop :=
  | red_appabs: ∀ t a v,
      isvalue v →
      red (App (Fun t a) v) (vsubst_term a 0 v)
  | red_tapptabs: ∀ t a t',
      red (TApp (TFun t a) t') (vsubst_tety a 0 t')
  | red_appfun: ∀ a a' b,
      red a a' → red (App a b) (App a' b)
  | red_apparg: ∀ v b b',
      isvalue v → red b b' → red (App v b) (App v b')
  | red_tapp: ∀ a a' t,
      red a a' → red (TApp a t) (TApp a' t).
```

We now give an alternate specification of the reduction relation in the style of Wright and Felleisen. The *red_top* relation captures beta-reductions at the top of a term. Reductions within terms are expressed using reduction contexts (see the *red_context* relation). Contexts are represented as functions from terms to terms whose shape is constrained by the *is_context* predicate.

```
Inductive red_top: term → term → Prop :=
  | red_top_appabs: ∀ t a v,
      isvalue v →
      red_top (App (Fun t a) v) (vsubst_term a 0 v)
  | red_top_tapptabs: ∀ t a t',
      red_top (TApp (TFun t a) t') (vsubst_tety a 0 t').

Inductive is_context: (term → term) → Prop :=
  | iscontext_hole:
      is_context (fun a ⇒ a)
  | iscontext_app_left: ∀ c b,
      is_context c → is_context (fun x ⇒ App (c x) b)
  | iscontext_app_right: ∀ v c,
      isvalue v → is_context c → is_context (fun x ⇒ App v (c x))
  | iscontext_tapp: ∀ c t,
      is_context c → is_context (fun x ⇒ TApp (c x) t).

Inductive red_context: term → term → Prop :=
  | red_context_intro: ∀ a a' c,
      red_top a a' → is_context c → red_context (c a) (c a').
```

The Plotkin-style relation is more convenient for doing formal proofs. Since the challenge is given in terms of contexts, we feel obliged to prove the equivalence between the two formulations of reduction. The proofs are routine inductions over the derivations of *red* and *is_context*, respectively.

```
Lemma red_red_context: ∀ a a', red a a' → red_context a a'.

Lemma red_context_red: ∀ a a', red_context a a' → red a a'.
```

3.5 Type soundness proof

Type soundness for $F_{<:}$ is established by proving the standard properties of type preservation (also called subject reduction) and progress.

3.5.1 Preservation

Technical inversion lemmas on typing derivations. These lemmas are similar (but not fully identical) to lemma A.13 in the on-paper proof.

```
Lemma has_type_fun_inv:
  ∀ e a t, has_type e a t →
  ∀ b s1 u1 u2, a = Fun s1 b → is_subtype e t (Arrow u1 u2) →
  is_subtype e u1 s1 ∧
  ∃ s2,
    is_subtype e s2 u2 ∧
    (∀ x, kind x = TERM → ¬In x (dom e) → has_type ((x, s1) :: e) (freshen_term b x)
s2).
```

Lemma *has_type_tfun_inv*:

```
  ∀ e a t, has_type e a t →
  ∀ b s1 u1 u2, a = TFun s1 b → is_subtype e t (Forall u1 u2) →
  is_subtype e u1 s1 ∧
  ∃ s2,
    (∀ x, kind x = TYPE → ¬In x (dom e) →
        is_subtype ((x, u1) :: e) (freshen_type s2 x) (freshen_type u2 x)) ∧
    (∀ x, kind x = TYPE → ¬In x (dom e) →
        has_type ((x, s1) :: e) (freshen_tety b x) (freshen_type s2 x)).
```

The preservation theorem states that if term *a* reduces to *a'*, then all typings valid for *a* are also valid for *a'*. It is proved by an outer induction on the reduction and an inner induction on the typing derivation (to get rid of subtyping steps).

Theorem *preservation*: ∀ *e a a' t*, *red a a'* → *has_type e a t* → *has_type e a' t*.
Proof.

```
  assert (∀ a a', red a a' →
            ∀ e a0 t, has_type e a0 t → ∀ (EQ: a = a0),
            has_type e a' t).
  induction 1; induction 1; intros; simplify_eq EQ; clear EQ; intros; subst;
  try (eapply t_sub; eauto; fail).
Case app abs
  assert (is_subtype e (Arrow t1 t2) (Arrow t1 t2)). apply sub_refl; eauto.
  destruct (has_type_fun_inv _ _ _ H0_ _ _ _ _ (refl_equal _) H0)
  as [A [s2 [B C]]].
  apply t_sub with s2; auto.
  destruct (fresh_name TERM (dom e ++ fv_term a)) as [x [F K]].
  rewrite (vsubst_psubst_freshen_term x); eauto.
  change e with (nil ++ e).
  apply has_type_stable_term_subst with t; auto.
  apply t_sub with t1; auto.
  simpl; eauto.
Case tapp tabs
  assert (is_subtype e (Forall t1 t2) (Forall t1 t2)). apply sub_refl; eauto.
  destruct (has_type_tfun_inv _ _ _ H _ _ _ _ (refl_equal _) H1)
  as [A [s2 [B C]]].
  destruct (fresh_name TYPE (dom e ++ fv_term a ++ fv_type t2)) as [x [F K]].
  rewrite (vsubst_psubst_freshen_tety x); eauto.
```

  rewrite (*vsubst_psubst_freshen_type x*); *eauto.*
  apply t_sub with (*psubst_type* (*freshen_type s2 x*) *x t0*).
  change e with (*psubst_env nil x t0* ++ *e*).
  apply has_type_stable_type_subst with t; *eauto.*
  apply sub_trans with t1; *auto.*
  simpl; *auto.*
  change e with (*psubst_env nil x t0* ++ *e*).
  apply sub_stable_subst with t1; *eauto. simpl*; *auto.*

Case context left app
  apply t_app with t1; *eauto.*

Case context right app
  apply t_app with t1; *eauto.*

Case context left tapp
  apply t_tapp with t1; *eauto.*

Final conclusion
  eauto.

Qed.

3.5.2 Progress

The following lemma, which corresponds to lemma A.14 in the challenge statement, determines the shape of a value from its type. Namely, closed values of function types are function abstractions, and closed values of polymorphic types are type abstractions.

Lemma *canonical_forms*:
  ∀ *e a t*, *has_type e a t* → *e* = *nil* → *isvalue a* →
  match t with
  | *Arrow t1 t2* ⇒ ∃ *s*, ∃ *b*, *a* = *Fun s b*
  | *Forall t1 t2* ⇒ ∃ *s*, ∃ *b*, *a* = *TFun s b*
  | *Top* ⇒ *True*
  | _ ⇒ *False*
  end.

The progress theorem shows that a term well-typed in the empty environment is never "stuck": either it is a value, or it can reduce. The theorem is proved by a simple induction on the typing derivation for the term and a case analysis on whether the subterms of the term are values or can reduce further.

Theorem *progress*: ∀ *a t*, *has_type nil a t* → *isvalue a* ∨ ∃ *a'*, *red a a'*.
Proof.
  assert (∀ *e a t*, *has_type e a t* → *e* = *nil* → *isvalue a* ∨ *exists a'*, *red a a'*).
  induction 1; *intros*; *subst e.*

Free variable: impossible in the empty typing environment.

simpl in H1. discriminate.
Function: already a value.
left; *constructor.*
Application *App a b*.
right.
destruct (*IHhas_type1* (*refl_equal* _)) *as* [*Va* | [a' Ra]].
destruct (*IHhas_type2* (*refl_equal* _)) *as* [*Vb* | [b' Rb]].
a and *b* are values. *a* must be a *Fun*. Beta-reduction is possible.
generalize (*canonical_forms* _ _ _ *H* (*refl_equal* _) *Va*).
intros [*s* [c EQ]]. *subst a.*
exists (*vsubst_term c* 0 *b*). *constructor. auto.*
a is a value, but *b* reduces. *App a b* therefore reduces.
exists (*App a b'*). *constructor*; *auto.*
a reduces. *App a b* reduces as well.
exists (*App a' b*). *constructor*; *auto.*
Type abstraction: already a value.
left; *constructor.*
Type application *TApp a t*.
right. destruct (*IHhas_type* (*refl_equal* _)) *as* [*Va* | [a' Ra]].
a is a value. *a* must be a *TFun*. Beta-reduction is possible.
generalize (*canonical_forms* _ _ _ *H* (*refl_equal* _) *Va*).
intros [*s* [b EQ]]. *subst a.*
exists (*vsubst_tety b* 0 *t*). *constructor.*
a reduces, and so does *TApp a t*.
exists (*TApp a' t*). *constructor*; *auto.*
Subtyping step.
auto.
Final conclusion.
eauto.
Qed.

Chapter 4

Execution of the dynamic semantics

In this chapter, we consider the problem of executing $F_{<:}$ terms as prescribed by the reduction semantics for this language. Such executions are useful for testing that the semantics has the intended behavior. This goal is listed as part 3 in the POPLmark challenge. As we will see, our development will go one step further and result in the production of an efficient and provably correct interpreter for $F_{<:}$.

There are two approaches to executing dynamic semantics within Coq. The first operates directly on a relational specification of the semantics, either big-step or small-step like our *red* predicate from chapter 3. The *eauto* Coq tactic, which build proofs by Prolog-style resolution over a set of predeclared inference rules and lemmas, can be abused to search and build derivation trees for a goal of the form $\exists$ *b*, *red a b*, therefore executing one reduction step from *a*. An example of this approach can be found in our work with A. Appel on the list-machine benchmark [AL06]. However, this approach is tricky to set up and very inefficient.

The other approach, which we follow in this chapter, is to specify the operational semantics as functions rather than predicates. While Coq has no efficient built-in execution mechanism for logic programs (composed of inductively-defined predicates), it can natively evaluate functional programs (composed of functions defined by recursion and pattern-matching). Such functional reductions are actually part of the logic of Coq, via the notion of conversion.

We therefore proceed in two steps. We will first define functions that compute the one-step or N-step reduct of a $F_{<:}$ term, and prove that they are correct and complete with respect to the relational semantics. We will then use these functions to evaluate terms within Coq and to extract efficient Caml code for an interpreter.

Require Import *Arith*.
Require Import *ZArith*.

```
Require Import List.
Require Import extralibrary.
Require Import part1a.
Require Import part2a.
```

4.1 Execution of one-step reductions

We first show that the *isvalue* predicate is decidable. The lemma below will actually provides us with a decision procedure that takes any term *a* and returns whether it is a value or not. We can then use this decision procedure within function definitions.

```
Lemma isvalue_dec:
  ∀ a, {isvalue a} + {~isvalue a}.
```

The *reduce* function maps a term *a* to either *Some b* if *a* reduces in one step to *b*, or to *None* if *a* does not reduce. It is defined by structural recursion over *a* and case analysis on whether subterms of *a* are values, or reduce, or are stuck.

```
Fixpoint reduce (a: term) : option term :=
  match a with
  | App b c ⇒
      if isvalue_dec b then
        if isvalue_dec c then
          match b with Fun t d ⇒ Some (vsubst_term d 0 c) | _ ⇒ None end
        else
          match reduce c with Some c' ⇒ Some(App b c') | None ⇒ None end
      else
        match reduce b with Some b' ⇒ Some(App b' c) | None ⇒ None end
  | TApp b t ⇒
      if isvalue_dec b then
        match b with TFun t' c ⇒ Some (vsubst_tety c 0 t) | _ ⇒ None end
      else
        match reduce b with Some b' ⇒ Some(TApp b' t) | None ⇒ None end
  | _ ⇒ None
  end.
```

We then show that this function is correct and complete with respect to the reduction rules: *reduce a* = *Some b* if and only if *red a b* holds. The proofs are routine inductions on the structure of *a* for the "only if" part and on the derivation of *red a b* for the "if" part.

```
Lemma reduce_is_correct:
  ∀ a a', reduce a = Some a' → red a a'.

Lemma isvalue_dec_true:
  ∀ a (T: Set) (b c: T), isvalue a → (if isvalue_dec a then b else c) = b.
Lemma isvalue_dec_false:
  ∀ a a' (T: Set) (b c: T), red a a' → (if isvalue_dec a then b else c) = c.
Lemma reduce_is_complete:
  ∀ a a', red a a' → reduce a = Some a'.
```

4.2 Execution of N-step reductions

The following function iterates the one-step reduction function *compute* to obtain the normal form of a term. Since Coq functions must always terminate, we need to bound the number of iterations by the n parameter. If a normal form cannot be reached in n steps, *compute* returns *None*.

```
Fixpoint compute (n: nat) (a: term) {struct n}: option term :=
  match n with
  | O ⇒ None
  | S n' ⇒
      match reduce a with
      | Some a' ⇒ compute n' a'
      | None ⇒ Some a
      end
  end.
```

We now show that *compute a*, if it succeeds, returns a reduct of a that is in normal form (irreducible).

```
Definition irreducible (a: term): Prop := ∀ b, ¬red a b.
Inductive red_sequence: term → term → Prop :=
  | red_sequence_0:
      ∀ a, irreducible a → red_sequence a a
  | red_sequence_1: ∀ a b c,
      red a b → red_sequence b c → red_sequence a c.
Lemma compute_correct:
  ∀ n a a', compute n a = Some a' → red_sequence a a'.
```

Conversely, if a term a has a normal form *a'*, there exists a number of iterations n such that *compute* returns *Some a'*.

```
Lemma compute_complete:
  ∀ a a', red_sequence a a' → ∃ n, compute n a = Some a'.
```

4.3 Experiments

We can now use the Coq directives *Eval compute in* (*reduce a*) and *Eval compute in* (*compute N a*) to display the results of performing one or N reduction steps in a.

```
Definition F_poly_identity := TFun Top (Fun (Tvar 0) (Var 0)).
Definition F_top_identity := TApp F_poly_identity Top.
Definition F_delta := Fun (Arrow Top Top) (App (Var 0) (Var 0)).
Definition F_testprog := App F_delta F_top_identity.

Eval compute in (reduce F_testprog).
Eval compute in (compute 100 F_testprog).
```

The latter returns *Some* (*Fun Top* (*Var* 0)), which is indeed the value of the term *F_testprog*. For a larger example, here is some arithmetic on Church integers.

```
Definition F_one : term :=
  (TFun Top (TFun (Tvar 0) (TFun (Tvar 1)
        (Fun (Arrow (Tvar 2) (Tvar 1))
          (Fun (Tvar 0)
            (App (Var 1) (Var 0))))))).

Definition F_nat : type :=
  (Forall Top
    (Forall (Tvar 0)
      (Forall (Tvar 1)
        (Arrow (Arrow (Tvar 2) (Tvar 1)) (Arrow (Tvar 0) (Tvar 1)))))).

Definition F_add : term :=
  (Fun F_nat
    (Fun F_nat
      (TFun Top (TFun (Tvar 0) (TFun (Tvar 1)
         (Fun (Arrow (Tvar 2) (Tvar 1))
           (Fun (Tvar 0)
              (App (TApp (TApp (TApp (Var 3) (Tvar 2)) (Tvar 1)) (Tvar 1))
                   (App (Var 1)
                        (App (TApp (TApp (TApp (Var 2) (Tvar 2)) (Tvar 1))
                                   (Tvar 0))
                             (App (Var 1) (Var 0))))))))))).

Eval compute in (compute 100 (App (App F_add F_one) F_one)).
```

Execution is nearly instantaneous. In Coq 8.1, we can also use *Eval vm_compute* to request evaluation via compilation to virtual machine code. This results in execution speed comparable to that of bytecoded OCaml.

An alternate execution path is to generate (or “extract” in Coq’s terminology) Caml code from the Coq definition of function *compute*. This is achieved by the following command:

```
Extraction "/tmp/fsub_eval.ml" compute.
```

The generated Caml code can be compiled with the OCaml native-code compiler for even higher execution speed. More importantly, it can be linked with a lexer, parser and printer hand-written in OCaml, obtaining a stand-alone reference interpreter for $F_{<:}$ that can execute non-trivial programs.

Chapter 5

Assessment

The POPLmark challenge is still ongoing, and the many solutions submitted differ widely — in the encodings of binders used, in the proof assistants used, and even in the proof styles of each author. It is therefore too early to draw conclusions. In this final chapter, we try to assess the quality of our solution with respect to various criteria.

Legibility of definitions and theorems Mechanized proofs often define notions and state theorems in ways that are somewhat different from what a mathematician would do in an on-paper proof. Often, the mechanized definitions and statements are more precise, but also harder to read and to relate with one's intuitions. The locally nameless approach followed in this report remains quite close to the on-paper definitions and theorems given in the statement of the POPLmark challenge. Definitions and statements are mostly unsurprising, except perhaps for the cases of definitions that involve crossing a binder. For instance, definition of subtyping between $\forall$ types is written on paper as

$$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \Gamma,\ X <: \tau_1 \vdash \sigma_2 <: \tau_2}{\Gamma \vdash (\forall X <: \sigma_1.\ \sigma_2) <: (\forall X <: \tau_1.\ \tau_2)}$$

while in our approach it is written as

$$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \forall X,\ X \notin \mathrm{Dom}(\Gamma) \Longrightarrow \Gamma,\ X <: \tau_1 \vdash \sigma_2[0 \leftarrow X] <: \tau_2[0 \leftarrow X]}{\Gamma \vdash (\forall <: \sigma_1.\ \sigma_2) <: (\forall <: \tau_1.\ \tau_2)}$$

There are several "tricks" here that surprise the reader: the de Bruijn notation for the bound variable, the substitutions $[0 \leftarrow X]$ in the second premise, and the placement of the quantification over X. In contrast, a purely nominal approach such as the solution by Urban *et al.* leads to a less mysterious definition of the form

$$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad X \text{ fresh for } \Gamma \qquad \Gamma,\ X <: \tau_1 \vdash \sigma_2 <: \tau_2}{\Gamma \vdash (\forall X <: \sigma_1.\ \sigma_2) <: (\forall X <: \tau_1.\ \tau_2)}$$

Nonetheless, we believe that definitions and theorems written in the locally nameless style are globally more readable and intuitively understandable than their equivalents using either de Bruijn indices or higher-order abstract syntax.

Overheads Mechanized proofs must make explicit a great number of small details and obvious properties that are omitted in research papers. In the case of type systems and operational semantics, this includes basic properties of terms, binders (α-conversion), substitutions, typing environments, and reduction sequences. All these properties had to be painfully made explicit in the present development. We estimate that such scaffolding work represent more than half of our development. More precisely, we consider that the following parts are pure overhead, compared with a paper proof:

- Definition and properties of free variables.
- Definition and properties of substitutions. There are two substitutions per data type of interest (one for free names and the other for bound de Bruijn variables), with the associated commutation properties.
- Definition and properties of swaps.
- Proving equivariance for many definitions.
- Proving admissibility of the rules with $\exists X \ldots$ in the premises, instead of $\forall X \ldots$
- Manipulations of environments, e.g. properties of environments of the form $\Gamma_1,\ X <: \tau,\ \Gamma_2$.

A legitimate question to ask is: how much of this overhead could be avoided, either by factoring the definitions and properties in reusable libraries, or by generating them automatically from high-level descriptions of the syntax of terms? Examples of reusable libraries include Urban and Tasson's nominal package, based on Isabelle/HOL type classes [UT05], and Chlipala's library for typing environments. However, such forms of reuse are limited in Coq by lack of type classes and any other form of "polytypic" definitions. Automatic generation in the style of Pottier's Cαml [Pot06] is another direction that remains to be investigated.

Size of the development Our solution is neither particularly compact nor excessively verbose compared with other solutions. Arthur Charguéraud compared the sizes of several Coq solutions to part 1A of the challenge, counting the number of non-trivial tactics invoked. The results are as follows:

Authors	Tactics	Representation used
Jérome Vouillon	431	de Bruijn indices
Aaron Stump	1147	names and levels
Xavier Leroy	630	locally nameless
Hirschowitz & Maggesi	1615	de Bruijn indices
Adam Chlipala	342	locally nameless
Arthur Charguéraud	233	locally nameless

Several factors contribute to this relative verbosity of our solution. First, as we discussed above, the locally nameless approach comes with inherent overheads. Second, as pointed out by Charguéraud, there are small variations of the locally nameless approach that can reduce the proof effort significantly (see below). Finally, unlike Vouillon, Chlipala and Charguéraud, we did not really take advantage of Coq's facilities for defining domain-specific tactics.

Possible improvements to the locally nameless approach Charguéraud [Cha06] and Charguéraud, Pierce and Weirich [CPW06] present several variations on the locally nameless approach that could significantly simplify our development. The most important one is to define predicates in the binder-crossing by quantification over all names not in a given finite set L of names, rather than all fresh names:

$$\forall L,\quad \frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \forall X,\ X \notin L \Longrightarrow \Gamma,\ X <: \tau_1 \vdash \sigma_2[0 \leftarrow X] <: \tau_2[0 \leftarrow X]}{\Gamma \vdash (\forall <: \sigma_1.\ \sigma_2) <: (\forall <: \tau_1.\ \tau_2)}$$

With this simple device, it is no longer necessary to prove equivariance of the well-formedness and subtyping relations. In turn, this removes the need to define and reason about swaps of names. The downside of this approach is reduced legibility: it becomes even less obvious why this rule captures the correct notion of subtyping between quantified types.

The second simplification is to treat well-formed typing environments as unordered sets of bindings, rather than ordered lists. For instance, instead of reasoning over environments of the form $\Gamma_1,\ \Gamma_2$, it is easier and often sufficient to reason over environments Γ that contain all bindings from Γ_1 and all bindings from Γ_2. Similarly, well-formedness of terms and types can profitably be defined with respect to a set of names bound in an environment rather than with respect to an environment.

Finally, the definition of the `vsubst` substitutions (replacement of a bound variable by a term) can also be simplified: instead of taking

$$\texttt{vsubst\_type}\ (\texttt{Tvar}\ n)\ x\ b = \begin{cases} \texttt{Tvar}\ n, & \text{if } n < x \\ b, & \text{if } n = x \\ \texttt{Tvar}\ (n-1), & \text{if } n > x \end{cases}$$

it suffices to take

$$\texttt{vsubst\_type}\ (\texttt{Tvar}\ n)\ x\ b = \begin{cases} \texttt{Tvar}\ n, & \text{if } n \neq x \\ b, & \text{if } n = x \end{cases}$$

since the case $n > x$ is never exercised in the rest of the development. This simple change significantly simplifies the proofs of the commutation lemmas between `psubst` and `vsubst` substitutions.

Executability of the semantics Part 3 of the POPLmark challenge (testing and animating the dynamic semantics) did not receive much attention from the participants. However, we believe that it is generally important to generate correct-by-construction implementations of programming tools from high-level specifications. The Coq proof assistant takes an almost schizophrenic stance on this issue. On the one hand, Coq provides excellent execution facilities for definitions written in functional style, both via its built-in evaluator (using an efficient virtual machine since version 8.1) and via its code extraction facility. In other work [Ler06, BDL06], we developed a whole optimizing C compiler by extraction from Coq functional specifications. On the other hand, Coq supports poorly the execution of specifications written in relational style (inductive predicates), forcing users to manually write functional variants of their relational specifications and to prove the equivalence of the functional and relational specifications, as we did in chapter 4. This is an annoyance in general, but also an interesting feature in some cases. In particular, it encourages a style where relational specifications are not polluted by executability constraints, and can e.g. be non-deterministic or even undecidable; executability constraints can then be taken into account later, in typical refinement style, when developing the functional specification.

Extension to records and record types The POPLmark challenge suggests to extend system $F_{<:}$ with records and record types with depth and width subtyping (parts 1B and 2B of the challenge). We considered briefly this extension but did not pursue it. This extension is conceptually rather easy but technically difficult: as in the other solutions that addressed parts 1B and 2B of the challenge, it appears necessary to define types and record types, as well as terms and record terms, in a mutually recursive manner. Such mutually recursive definitions are quite painful to handle in Coq, and require extensive changes to the proof scripts. Again, this is more a limitation of Coq than a fundamental difficulty.

Bibliography

[ABF$^+$05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 2005.

[AL06] Andrew W. Appel and Xavier Leroy. A list-machine benchmark for mechanized metatheory. Research report 5914, INRIA, 2006.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer-Verlag, 2006.

[Cha06] Arthur Charguéraud. A comparison between concrete representations for bindings. In *Workshop on Mechanizing Metatheory*, 2006.

[Coq07] The Coq proof assistant. Software and documentation available on the Web, `http://coq.inria.fr/`, 1995–2007.

[CPW06] Arthur Charguéraud, Benjamin C. Pierce, and Stephanie Weirich. Proof engineering: Practical techniques for mechanized metatheory. Manuscript, available from `http://www.cis.upenn.edu/~bcpierce/papers/binders.pdf`, September 2006.

[Gor94] Andrew D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *Higher-order logic theorem proving and its applications 1993*, volume 780 of *Lecture Notes in Computer Science*, pages 414–426. Springer-Verlag, 1994.

[Hue94] Gérard Huet. Residual theory in lambda-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[Ler07] Xavier Leroy. A locally nameless solution to the POPLmark challenge – the Coq development. `http://gallium.inria.fr/~xleroy/POPLmark/locally-nameless`, 2007.

[MM04] Conor McBride and James McKinna. I am not a number; I am a free variable. In *Proc. 2004 Haskell Workshop*. ACM Press, 2004.

[MP99] James McKinna and Randy Pollack. Some lambda calculus and type theory formalized. *J. Automated Reasoning*, 23:373–409, 1999.

[Pit03] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[Pot06] François Pottier. An overview of Cαml. In *ACM Workshop on ML*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 27–52, March 2006.

[UT05] Christian Urban and Christine Tasson. Nominal reasoning techniques in Isabelle/HOL. In *Proc. Int. Conf. on Automated Deduction (CADE)*, volume 3632 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, 2005.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399