Functional programming languages

Part VI: towards Coq mechanization

Xavier Leroy

INRIA Paris

MPRI 2-4, 2016-2017

Why formalize programming languages?

To obtain mathematically-precise definitions of languages.
(Dynamic semantics, type systems, . . .)

To obtain mathematical evidence of soundness for tools such as

- type systems (well-typed programs do not go wrong)
- type checkers and type inference
- static analyzers (e.g. abstract interpreters)
- program logics (e.g. Hoare logic, separation logic)
- deductive program provers (e.g. verification condition generators)
- interpreters
- compilers.

Challenge 1: scaling up

From calculi (λ, π) and toy languages (IMP, MiniML) to real-world languages (in all their ugliness), e.g. Java, C, JavaScript.

From abstract machines to optimizing, multi-pass compilers producing code for real processors.

From textbook abstract interpreters to scalable and precise static analyzers such as Astrée.

Challenge 2: trusting the maths

Authors struggle with huge LaTeX documents.

Reviewers give up on checking huge but rather boring proofs.

> *Proofs written by computer scientists are boring:*
> *they read as if the author is programming the reader.*
>
> *(John C. Mitchell)*

Few opportunities for reuse and incremental extension of earlier work.

Opportunity: machine-assisted proofs

Mechanized theorem proving has made great progress in the last 20 years. (Cf. the monumental theorems of Gonthier et al: 4 colors, Feit-Thompson.)

Emergence of engineering principles for large mechanized proofs.

The kind of proofs used in programming language theory are a good match for theorem provers:

- large definitions, many cases
- mostly syntactic techniques, no deep mathematics concepts.

The POPLmark challenge

In 2005, Aydemir et al challenged the POPL community:

> *How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?*

12 years later, about 20% of the papers at recent POPL conferences come with such an electronic appendix.

Proof assistants

1. A formal specification language to write definitions and state theorems.
2. Commands to build proofs, often in interaction with the user + some proof automation.
3. Often, an independent, automatic checker for the proofs thus built.

Popular proof assistants in programming language research:
Coq, HOL4, Isabelle/HOL.

Outline

Examples of Coq mechanization

See the commented Coq development on the course Web site.

Module	Contents	From lecture
`Sequences`	Library on sequences of transitions	
`Semantics`	Small-step and big-step semantics; equivalence proofs	Leroy's lecture #1
`Typing`	Simply-typed λ-calculus and type soundness proof	Rémy's lecture #1
`Compiler`	Compilation to Modern SECD and correctness proof	Leroy's lecture #2
`CPS`	CPS conversion and correctness proof	Leroy's lecture #3

Coq in a nutshell, 1: Computations and functions

A pure functional language in the style of ML, with recursive functions defined by pattern-matching.

```
Fixpoint factorial (n: nat) :=
  match n with
  | O   => 1
  | S p => n * factorial p
  end.

Fixpoint concat (A: Type) (l1 l2: list A) :=
  match l1 with
  | nil => l2
  | h :: t => h :: concat t l2
  end.
```

Note: all functions must terminate.

Coq in a nutshell, 2: mathematical logic

The usual logical connectors and quantifiers.

```
Definition divides (a b: N) := exists n: N, b = n * a.

Theorem factorial_divisors:
  forall n i, 1 <= i <= n -> divides i (factorial n).

Definition prime (p: N) :=
  p > 1 /\ (forall d, divides d p -> d = 1 \/ d = p).

Theorem Euclid:
  forall n, exists p, p >= n /\ prime p.
```

Coq in a nutshell, 3: inductive types

Like Generalized Abstract Data Types in OCaml and Haskell.

```
Inductive nat: Type :=
| O: nat
| S: nat -> nat.

Inductive list: Type -> Type :=
| nil: forall A, list A
| cons: forall A, A -> list A -> list A.
```

Coq in a nutshell, 4: inductive predicates

Similar to definitions by axioms and inference rules.

```
Inductive even: nat -> Prop :=
  | even_zero:
      even 0
  | even_plus_2:
      forall n, even n -> even (S (S n)).
```

Compare with the inference rules:

$$\texttt{even}\ 0 \qquad\qquad \frac{\texttt{even}\ n}{\texttt{even}\ (\texttt{S}(\texttt{S}(n)))}$$

Technically: `even` is the GADT that describes derivations of `even` n statements.

Lessons learned

Definitions and statements of theorems are close to what we did on paper.

Proof scripts are ugly but no one has to read them.

Ratio specs : proof scripts is roughly 1 : 1.

No alpha-conversion? (much more on this later).

Good reuse potential, for small extensions (e.g. primitive `let`, booleans, etc) and not so small ones (e.g. subtyping).

Outline

Bound variables and alpha-conversion

Most programming languages provide constructs that bind variables:

- Function abstractions $\lambda x.a$
- Definitions $\texttt{let}\ x = a\ \texttt{in}\ b$
- Pattern-matching $\texttt{match}\ a\ \texttt{with}\ (x, y) \to b$
- Quantifiers (in types) $\forall\alpha.\ \alpha \to \alpha$

It is customary to identify terms up to alpha-conversion, that is, renaming of bound variables:

$$\lambda x.\ x + 1 \equiv_\alpha \lambda y.\ y + 1 \qquad \forall\alpha.\ \alpha\ \texttt{list} \equiv_\alpha \forall\beta.\ \beta\ \texttt{list}$$

Substitutions, capture, and alpha-conversion

In the presence of binders, textual substitution $a\{x \leftarrow b\}$ must avoid capturing a free variable of b by a binder in a:

$$\begin{aligned} (\lambda y.\ x + y)\{x \leftarrow 2 \times z\} &= \lambda y.\ 2 \times z + y \quad \checkmark \\ (\lambda y.\ x + y)\{x \leftarrow 2 \times y\} &= \lambda y.\ 2 \times y + y \quad \times \end{aligned}$$

In the second case, the free y is captured by λy.

A major reason for considering terms up to alpha-conversion is that capture can always be avoided by renaming bound variables on the fly during substitution:

$$(\lambda y.\ x + y)\{x \leftarrow 2 \times y\} = (\lambda z.\ x + z)\{x \leftarrow 2 \times y\} = \lambda z.\ 2 \times y + z \quad \checkmark$$

This is called capture-avoiding substitution.

An oddity: no alpha-conversion ?!?

The Coq development uses names for free and bound variables, but terms are not identified up to renaming of bound variables.

$$\texttt{Fun(x, Var x)} \quad \neq \quad \texttt{Fun(y, Var y)} \qquad \texttt{if x} \neq \texttt{y}$$

Likewise, substitution is not capture-avoiding:

$$(\texttt{Fun}(y, a))\{x \leftarrow b\} = \texttt{Fun}(y, a\{x \leftarrow b\}) \quad \text{if } x \neq y$$

Correct only if b is closed.

To alpha-convert or not to alpha-convert?

α-conversion not needed	α-conversion required
For semantics	
Weak reductions of closed programs	Strong reductions Symbolic evaluation
For type systems	
Simply-typed Hindley-Milner (barely)	Polymorphism in general ($\forall$, $\exists$) System F and above. Dependent types.
For program transformations	
Compilation to abstract mach. Naive CPS conversion (barely) Naive monadic conversion	Compile-time reductions One-pass CPS conversion Administrative reductions

What is so hard with alpha-conversion?

Working with a quotient set:

$$\lambda x.a \equiv_\alpha \lambda y.\ a\{x \leftarrow y\} \quad \text{if } y \text{ not free in } a$$

Need to ensure that every definition and statement is compatible with this equivalence relation.

Example: the free variables of a term are defined by recursion on a representative of an equivalence class:

$$\begin{aligned} FV(x) &= x \\ FV(\lambda x.a) &= FV(a) \setminus \{x\} \\ FV(a\ b) &= FV(a) \cup FV(b) \end{aligned}$$

Must prove compatibility: $FV(a) = FV(b)$ if $a \equiv_\alpha b$.

What is so hard with alpha-conversion?

Need to define appropriate induction principles:

To prove a property P of a term by induction on the term, in the case $P(\lambda x.a)$, what can I assume as induction hypothesis?

- just that $P(a)$ holds?
- or that $P(a\{x \leftarrow y\})$ also holds for any y such that $\lambda x.a \equiv_\alpha \lambda y.a\{x \leftarrow y\}$?

What is so hard with alpha-conversion?

Just defining capture-avoiding substitution:

$$(\lambda y.a)\{x \leftarrow b\} = \lambda z.\ (a\{y \leftarrow z\})\ \{x \leftarrow b\} \quad \text{with } z \text{ fresh}$$

Need to define what "z fresh" means, e.g.
z is the smallest variable not free in a nor in b.

Then, we get a recursion that is not structural ($a\{y \leftarrow z\}$ is not a syntactic subterm of $\lambda y.a$) and therefore not directly accepted by Coq.

Must use a different style of recursion, e.g. on the size of the term rather than on its structure.

Mechanizing alpha-conversion

Several approaches to the handling of binders and alpha-conversion using interactive theorem provers. Described next:

1. de Bruijn indices
2. higher-order abstract syntax
3. nominal logics
4. locally nameless.

The "POPLmark challenge" compare these approaches (and more) on a challenge problem (type soundness for $F_{<:}$).

15 solutions in 5 proof assistants. No consensus.

Outline

de Bruijn indices

(Nicolaas de Bruijn, the Automath prover, 1967)

$$\begin{array}{lll} a ::= n & & \text{variable } (n \in \mathbf{N}) \\ \quad \mid \lambda.a & & \text{abstraction, binds variable 0} \\ \quad \mid a_1\ a_2 & & \text{application} \end{array}$$

Represent every variable by its distance to its binder.

$$\lambda x.\lambda y.\ x\ y \quad \equiv \quad \lambda.\lambda.\ 1\ 0 \quad \equiv \quad \lambda z.\lambda w.\ z\ w$$

A canonical representation: α-convertible terms are equal.

Used in many early mechanizations (e.g. G. Huet, *Residual Theory in lambda-Calculus: A Formal Development*, J. Funct. Program. 4(3), 1994; B. Barras and B. Werner, *Coq in Coq*, 1997).

Substitution with de Bruijn indices

The definition of substitution is not obvious:

$$
\begin{aligned}
x\{n \leftarrow a\} &= \begin{cases} x & \text{if } x < n \\ a & \text{if } x = n \\ x-1 & \text{if } x > n \end{cases} \\
(\lambda.b)\{n \leftarrow a\} &= \lambda.\ b\{n+1 \leftarrow \Uparrow_0 a\} \\
(b\ c)\{n \leftarrow a\} &= b\{n \leftarrow a\}\ c\{n \leftarrow a\}
\end{aligned}
$$

For abstractions, the indices of free variables in a must be incremented by one to avoid capturing variable 0 bound by the λ. This is achieved by the lifting operator $\Uparrow_n$, which increments all variables $\geq n$.

$$
\begin{aligned}
\Uparrow_n x &= \begin{cases} x & \text{if } x < n \\ x+1 & \text{if } x \geq n \end{cases} \\
\Uparrow_n \lambda.a &= \lambda.\ \Uparrow_{n+1} a \\
\Uparrow_n (a\ b) &= (\Uparrow_n a)\ (\Uparrow_n b)
\end{aligned}
$$

Properties of substitution and lifting

Some technical commutation lemmas: if $p \geq n$,

$$
\begin{aligned}
(\Uparrow_n a)\{n \leftarrow b\} &= a \\
\Uparrow_n (a\{p \leftarrow b\}) &= (\Uparrow_n a)\{p+1 \leftarrow \Uparrow_n b\} \\
\Uparrow_p (a\{n \leftarrow b\}) &= (\Uparrow_{p+1} a)\{n \leftarrow \Uparrow_p b\} \\
a\{n \leftarrow b\}\{p \leftarrow c\} &= a\{p+1 \leftarrow \Uparrow_n c\}\{n \leftarrow b\{p \leftarrow c\}\}
\end{aligned}
$$

Pro: systematic proofs, easy to automate.

Cons: not intuitive, unclear what indices really mean.

Issues with de Bruijn indices

Statements of theorems are polluted by adjustments of indices for free variables and don't look quite like what we're used to.

For example, in textbook, named notation, the weakening lemma is:

$$\textit{Weakening}: \quad E_1, E_2 \vdash a : \tau \implies E_1, x : \tau', E_2 \vdash a : \tau$$

with the same a in hypothesis and conclusion. (Plus: implicit hypothesis that x is not bound in E_1 or E_2.)

With de Bruijn indices, the second a is lifted by an amount that is equal to the number of binders after that of $\mathtt{x}$, namely, the length of the E_1 environment:

$$\textit{Weakening}: \quad E_1, E_2 \vdash a : \tau \implies E_1, \tau', E_2 \vdash \Uparrow_{|E_1|} a : \tau$$

Issues with de Bruijn indices

Likewise, for stability of typing by substitution, the top-level statement is quite readable:

$$\tau', E \vdash a : \tau \ \wedge \ E \vdash b : \tau' \implies E \vdash a\{0 \leftarrow b\} : \tau$$

but the statement that can be proved by induction is less intuitive:

$$E_1, \tau', E_2 \vdash a : \tau \ \wedge \ E_1, E_2 \vdash b : \tau' \implies E_1, E_2 \vdash a\{|E_1| \leftarrow b\} : \tau$$

Summary

de Bruijn indices make it possible to mechanize the theory of many languages, including difficult features such as

- multiple kinds of variables (e.g. type variables and term variables in System F)
- binders that bind multiple variables (e.g. ML pattern-matching).

The statements of theorems look somewhat different from what we do on paper using names, and take time getting used to.

Some "boilerplate" (definitions and properties of substitutions and lifting) is necessary, but can be automated to some extent. See for instance F. Pottier's DBlib library, `https://github.com/fpottier/dblib`.

Outline

Higher-Order Abstract Syntax (HOAS)

(Pfenning, Elliott, Miller, Nadathur, 1987–1988; the Twelf logical framework)

Leverage the power of your logic: it has α-conversion built-in!

Represent binding in terms using functions of your logic/language.

```
type term =                    --->      type term =
  | Var of name
  | Lam of name * term                     | Lam of term -> term
  | App of term * term                     | App of term * term
```

$\lambda x.\lambda y.\ x\ y$ is represented as `Lam(fun x -> Lam(fun y -> App(x,y)))`.

Substitution is just function application!

```
let betared (Lam f) arg = f arg
```

Higher-Order Abstract Syntax (HOAS)

How to reason about non-closed terms? Using logical implications and hypothetical judgments!

Example: typing rules for simply-typed λ-calculus

$$\frac{\mathtt{assm}(x, \tau)}{\vdash x : \tau} \qquad \frac{\forall x,\ (\mathtt{assm}(x, \tau_1) \Rightarrow\ \ \vdash f\ x : \tau_2)}{\vdash \mathtt{Lam}\ f : \tau_1 \to \tau_2}$$

The typing environment is encoded as the set of logical assumptions $\mathtt{assm}(x, \tau)$ available in the logical environment.

Why no HOAS in Coq?

Problem 1: "exotic" function terms.

In Coq or Caml, the type `term -> term` contains more functions than just HOAS encodings of λ-terms. The latter are parametric in their arguments, while Coq/Caml functions can also discriminate over their arguments, e.g.

```
f =  fun x -> match x with App(_,_) -> x | Lam _ -> App(x,x)
```

These "exotic" functions invalidate some expected properties of substitutions. For example, we expect

$$a\{x \leftarrow b\} = a\{x \leftarrow c\} \implies b = c \text{ or } \forall b, a\{x \leftarrow b\} = a$$

However, this property fails with the "exotic" function `f` above,

```
f (Lam g) = App(Lam g, Lam g) = f (App (Lam g, Lam g))
```

Why no HOAS in Coq?

Problem 2: contravariant inductive definitions are not allowed in Coq.

```
Inductive term := Lam : (term -> term) -> term
```
✘

It would make it possible to define nonterminating computations:

```
Definition delta (t: term): term :=
  match t with Lam f => f t end.
Definition omega : term := delta (Lam delta).
```

Likewise, inductive predicates with contravariant (negative) occurrences lead to logical inconsistency:

```
Inductive bot : Prop := Bot : (bot -> False) -> bot.
```
✘

By inversion, `bot` implies `bot -> False` and therefore `False`.

The Twelf system

(F, Pfenning, C. Schurmann et al; `http://twelf.plparty.org/`)

A logical framework that supports defining and reasoning upon languages and logics using Higher-Order Abstract Syntax:

- Terms of the language are encoded in LF using HOAS for binders (cf. lecture #3 of Y. Régis-Gianas).
- Properties of terms (such as well-typedness) and meta-properties (such as soundness of typing) are specified as inference rules in λ-Prolog.

The Coq difficulties with HOAS (exotic functions, nontermination) are avoided by restricting the expressiveness of the functional language (LF): it has no destructors (no pattern-matching), hence can only express functions that are "parametric" in their arguments.

Parametric HOAS

(Adam Chlipala, 2008)

A form of HOAS that is compatible with Coq and similar proof assistants.

Idea 1: break the contravariant recursion `term = Lam of term -> term` by introducing a type `V` standing for variables:

```
Inductive term (V: Type): Type :=
  | Var: V -> term V
  | Fun: (V -> term V) -> term V
  | App: term V -> term V -> term V.
```

Example: the identity function $\lambda x.x$ is represented by

```
Fun (fun (x: V) -> Var x)
```

for any type `V` of your choice.

Parametric HOAS

For a fixed type `V` (say, `nat`), `term V` still contains exotic functions:

```
Fun (fun n => match n with O => Var nat 1 | _ => Var nat n end)
```

Idea 2: rule out exotic functions by quantifying over all possible types `V`. Thus, the type of closed terms is

```
Definition term0 := forall V, term V.
```

and the type of terms having one free variable is

```
Definition term1 := forall V, V -> term V.
```

We can then define constructors for closed terms:

```
Definition APP (a b: term0) : term0 :=
  fun (V: Type) => App (a V) (b V).
Definition FUN (a: term1) : term0 :=
  fun (V: Type) => Fun (a V).
```

Substitution in parametric HOAS

Substitution is almost function application:

```
Definition subst (f: term1) (a: term0) : term0 :=
  fun (V: Type) => squash (f (term V) (a V))
```

`squash` is the canonical injection from `term (term V)` into `term V` obtained by erasing the `Var` constructors:

```
Fixpoint squash (V: Type) (a: term (term V)) : term V :=
  match a with
  | Var x => x
  | Fun f => Fun (fun v => f (Var v))
  | App b c => App (squash b) (squash c)
  end.
```

For more information

Chapter 17 of *Certified Programming with Dependent Types*, A. Chlipala.

The Lambda Tamer library `http://ltamer.sourceforge.net/`

Outline

Nominal approaches

Nominal logic (A. Pitts, 2001):
A first-order logic that internalizes the notions of names, binding, and equivariance (invariance under α-conversion).

The Nominal package for Isabelle/HOL (Ch. Urban et al, 2006):
A library for the Isabelle/HOL proof assistant that implements the main notions of nominal logic and automates the definition of data types with binders (e.g. AST for programming languages) modulo α-conversion.

Example 1 (Nominal Isabelle definition)

```
atom_decl name
nominal_datatype lam = Var "name"
                     | App "lam × lam"
                     | Lam "≪name≫ lam"
```

Automatically defines the correct equality-modulo-α over type `lam`, as well as an appropriate induction principle.

Names and swaps

α-equivalence is defined not in terms of renamings $\{x \leftarrow y\}$ but in terms of swaps (permutations of two names)

$$\binom{x}{y} = \{x \leftarrow y; y \leftarrow x\}$$

Unlike renamings, swaps are bijective (self-inverse) and can be applied uniformly both to free and bound variables, with no risk of capture:

$$\binom{x}{y}(\lambda z.a) = \lambda \binom{x}{y}(z).\ \binom{x}{y}(a)$$

Nominal types

A nominal type is a type equipped with a swap operation satisfying common-sense properties such as

$$\binom{x}{y}\binom{x}{y}t = t \qquad \binom{\binom{x}{y}u}{\binom{x}{y}v}t = \binom{u}{v}\binom{x}{y}\binom{u}{v}t$$

Many types are nominal: the type of names; numbers and other constants; and the product, sum or list of nominal types.

Support and freshness

For any nominal type, we can define generic notions of

- Occurrence (a name is mentioned in a term)

$$x \in t \Leftrightarrow \binom{x}{y} t \neq t \text{ for infinitely many } y$$

- Support (all names mentioned in a term)

$$\mathrm{supp}(t) = \{x \mid x \in t\}$$

- Freshness (a name is not mentioned in a term)

$$x\#t \Leftrightarrow x \notin \mathrm{supp}(t) \Leftrightarrow \binom{x}{y} t = t \text{ for infinitely many } y$$

Binders

$[x].t$ represents the term t in which name x is bound (e.g. $\lambda x.t$ or $\forall x.t$).

It is characterized by:

$$[x].t = [x'].t' \quad \text{iff} \quad (x = x' \wedge t = t') \vee (x \neq x' \wedge t = \binom{x}{x'} t' \wedge x\#t')$$

$$y\#[x].t \quad \text{iff} \quad y \neq x \wedge y\#t$$

Binders $[x].t$ can be constructed as partial functions from names to t's:

$$\begin{aligned}[x].t = \lambda y.\ & \texttt{if } x = y \texttt{ then Some}(t) \\ & \texttt{else if } y\#t \texttt{ then Some}(\tbinom{x}{y} t) \\ & \texttt{else None}\end{aligned}$$

The freshness quantifier

If $\phi(x, \vec{y})$ is a formula of nominal logic involving a (morally fresh) name x and (morally free) names $\vec{y}$, then

$$\mathbf{N}x.\ \phi(x, \vec{y}) \stackrel{def}{=} \forall x \# \vec{y}.\ \phi(x, \vec{y}) \Leftrightarrow \exists x \# \vec{y}.\ \phi(x, \vec{y})$$

Proof: in Pitt's nominal logic, all formulas are invariant by swaps. Assume $\phi(x, \vec{y})$ for one $x \# \vec{y}$. Then, for every $x' \# \vec{y}$,

$$\binom{x}{x'} \phi(x, \vec{y}) = \phi(x', \vec{y}) \text{ holds}$$

In Nominal Isabelle/HOL, not all formulas are invariant by swaps, but this $\forall$-$\exists$ equivalence is built in the induction principles generated.

The freshness quantifier

Example of use: the typing rule for λ-abstraction.

$$\frac{\mathbf{N}x.\ E + \{x : \tau_1\} : a : \tau_2}{E \vdash \lambda x.a : \tau_1 \to \tau_2}$$

To conclude $E \vdash \lambda x.a : \tau_1 \to \tau_2$, it suffices to exhibit one sufficiently fresh x that satisfies $E + \{x : \tau_1\} : a : \tau_2$.

When we know that $E \vdash \lambda x.a : \tau_1 \to \tau_2$, we can assume $E + \{x : \tau_1\} : a : \tau_2$ for any sufficiently fresh x.

Outline

The locally nameless representation

(R. Pollack, circa 2004; A. Charguéraud et al, 2008)

A concrete representation where bound variables and free variables are syntactically distinct:

```
Inductive term :=
  | BVar: nat -> term          (* bound variable *)
  | FVar: name -> term         (* free variable *)
  | Fun: term -> term
  | App: term -> term -> term.
```

Bound variables = de Bruijn indices.
Free variables = names.

Example: $\lambda x.\ xy$ is `Fun (App (BVar 0) (FVar "y"))`.

Invariant: terms are locally closed (no free de Bruijn indices).

The locally nameless representation

Two substitution operations:
of a term for a name $\{x \leftarrow a\}$
of a term for a de Bruijn index $[n \leftarrow a]$.

$$(\lambda.a)\{x \leftarrow b\} = \lambda.\ a\{x \leftarrow b\} \qquad (\lambda.a)[n \leftarrow b] = \lambda.\ a[n+1 \leftarrow b]$$

No variable capture can occur:
in $[n \leftarrow b]$ because b is locally closed
in $\{x \leftarrow b\}$ because λ does not bind any name.

Some commutation properties must be proved, but fewer and simpler than for de Bruijn indices.

Working with binders

How to descend within a term $\lambda.a$?

- Bad: recurse on a (not locally closed!)
- Good: recurse on $a^x = a[0 \leftarrow x]$ for a fresh name x.

Example: the typing rule for λ-abstractions:

$$\frac{x \notin FV(E) \cup FV(a) \qquad E + \{x : \tau_1\} \vdash a^x : \tau_2}{E \vdash \lambda.a : \tau_1 \rightarrow \tau_2}$$

How to quantify freshness

$$\frac{\forall x, x \notin FV(E) \cup FV(a) \Rightarrow E + \{x : \tau_1\} \vdash a^x : \tau_2}{E \vdash \lambda.a : \tau_1 \to \tau_2} \text{ (universal)}$$

$$\frac{\exists x, x \notin FV(E) \cup FV(a) \wedge E + \{x : \tau_1\} \vdash a^x : \tau_2}{E \vdash \lambda.a : \tau_1 \to \tau_2} \text{ (existential)}$$

Universal quantification: the premise must hold for all fresh names.

- Maximally strong for elimination and induction: knowing $E \vdash \lambda.a : \tau_1 \to \tau_2$, we get infinitely many possible choices for x that represents the parameter in the premise.
- Inconvenient for introduction: to prove $E \vdash \lambda.a : \tau_1 \to \tau_2$ we must show the premise for all fresh x.

How to quantify freshness

$$\frac{\forall x, x \notin FV(E) \cup FV(a) \Rightarrow E + \{x : \tau_1\} \vdash a^x : \tau_2}{E \vdash \lambda.a : \tau_1 \to \tau_2} \text{ (universal)}$$

$$\frac{\exists x, x \notin FV(E) \cup FV(a) \wedge E + \{x : \tau_1\} \vdash a^x : \tau_2}{E \vdash \lambda.a : \tau_1 \to \tau_2} \text{ (existential)}$$

Existential quantification: the premise must hold for one fresh name.

- Very convenient for introduction: to prove $E \vdash \lambda.a : \tau_1 \to \tau_2$ it suffices to show the premise for one x.
- Weak for elimination and induction: knowing $E \vdash \lambda.a : \tau_1 \to \tau_2$, we get the premise for only one x that we cannot choose and may not satisfy other freshness conditions coming from other parts of the proof.

Equivalence between the two quantifications

As in nominal logic, the two rules (universal) and (existential) are equivalent if the judgement $E \vdash a : \tau$ is equivariant, that is, stable under swaps of names:

$$E \vdash a : \tau \implies \begin{pmatrix} x \\ y \end{pmatrix} E \vdash \begin{pmatrix} x \\ y \end{pmatrix} a : \begin{pmatrix} x \\ y \end{pmatrix} \tau$$

Equivalence between the two quantifications

R. Pollack and others take the (universal) rule as the definition (thus obtaining a strong induction principle), then show the (existential) rule as a theorem:

```
Inductive hastype: env -> term -> type -> Prop :=
  ...
  | hastyp_abstr: forall E a t1 t2,
      (forall x, ~In x (FV E) -> ~In x (FV a) ->
                 hastype ((x, t1) :: E) (open x a) t2) ->
      hastype E (Lam a) (Arrow t1 t2).

Lemma hastyp_abstr_weak:
  forall E a t1 t2 x,
  ~In x (FV E) -> ~In x (FV a) ->
  hastype ((x, t1) :: E) (open x a) t2 ->
  hastype E (Lam a) (Arrow t1 t2).
```

Problem: must prove equivariance for all definitions.

Cofinite quantification

In their LN library `http://www.chargueraud.org/softs/ln/`, A. Charguéraud et al use a form of freshness quantification itermediate between the (universal) and (existential) forms: cofinite quantification.

$$\frac{\exists L, \forall x, x \notin L \Rightarrow E + \{x : \tau_1\} \vdash a^x : \tau_2}{E \vdash \lambda.a : \tau_1 \to \tau_2}\ \text{(cofinite)}$$

L is a finite set of names that must be "avoided". It typically includes $FV(E) \cup FV(a)$ but can be larger than that.

- For elimination and induction, we still have that the premise holds for infinitely many x.
- For introduction, we can choose L large enough to exclude all choices of x that might make it difficult to prove the premise.

Advantage: no need to prove equivariance for definitions.

Locally nameless at work

See the beautiful examples of use for the LN library,
`http://www.chargueraud.org/softs/ln/`

Some representative mechanized verifications

Authors	Topic	Prover	α?
	Type systems		
Barras & Werner	Coq in Coq	Coq	dB
Crary & Harper	Standard ML (intermediate language)	Twelf	HOAS
Charguéraud	Type soundness for $F_{<:}$, mini-ML, CoC	Coq	LN
Dubois & Menissier	Algorithm W	Coq	none
Nipkow	Algorithm W	Isabelle	none
Nipkow & Urban	Algorithm W	Isabelle	nominal
Pottier & Balabonski	The Mezzo language	Coq	dB
	Compilers		
Leroy et al	CompCert C $\rightarrow$ PowerPC/ARM/x86	Coq	none
Chlipala	mini-ML + exns + refs $\rightarrow$ asm	Coq	PHOAS
Dargaye	mini-ML $\rightarrow$ Cminor	Coq	dB
Klein & Nipkow	mini-Java $\rightarrow$ JVM + bytecode verif.	Isabelle	none
Myreen et al	Cake ML $\rightarrow$ VM $\rightarrow$ x86-64	HOL	none