Machine Assistance for Programming Language Research

Stephanie Weirich

Penn PL
University of Pennsylvania

Changes in PL Research

- Increasing complexity of language features being considered
  - E.g., module systems, dependently typed programming, types ensuring resource bounds or deadlock freedom, …
- Increasing concern for scale and integration
- Decreasing cycle time for tech transfer
  - Extensions to production languages
    - e.g., generics in Java/C#
  - Adoption of research languages
    - e.g. (OCaml, F#, Haskell, Scheme)

→ Big changes in the way *proofs* are communicated…

The State of the Art

Chen and Tarditi,
A Simple Typed Intermediate Language for Object-Oriented Languages,
Principles of Programming Languages (POPL), 2005

We have proved the soundness of LIL_C, in the style of [34], and the decidability of type checking. Full proofs are in the technical report.

THEOREM 1 (PRESERVATION). *If $\Sigma \vdash P : \tau$ and $P \mapsto P'$, then $\exists \Sigma'$ such that $\Sigma' \vdash P' : \tau$.*

THEOREM 2 (PROGRESS). *If $\Sigma \vdash P : \tau$, then either the main expression in P is a value, or $\exists P'$ such that $P \mapsto P'$.*

Proof sketch: by standard induction over the typing rules.

3. *If $\Theta;\bullet;\Sigma;\Gamma \vdash v : \forall tvs(\tau_1,\ldots,\tau_n) \to \tau$ ($tvs = \alpha_1 \ll u_1,\ldots,\alpha_m \ll u_m$) and $\Theta \vdash H : \Sigma$, then v is a label and $H(v) = fix\ g\langle tvs'\rangle(x_1 : \tau_1',\ldots,x_n : \tau_n') : \tau' = e_m$ and $tvs' = \alpha_1 \ll u_1',\ldots,\alpha_m \ll u_m'$.*

4. *If $\Theta;\bullet;\Sigma;\Gamma \vdash v : \exists\alpha \ll \tau_u.\ \tau$, then $v = pack\ \tau_0$ as $\alpha \ll \tau_u'$ in $(v' : \tau')$.*

5. *If $\Theta;\bullet;\Sigma;\Gamma \vdash v : Tag(\tau)$, then $v = tag(C)$ for some C.*

6. *If $\Theta;\bullet;\Sigma;\Gamma \vdash v : Tag(C)$, then $v = tag(C)$.*

7. *If $\Theta;\bullet;\Sigma;\Gamma \vdash v : C$, then $v = C(v')$ for some value v'.*

8. *If $\Theta;\bullet;\Sigma;\Gamma \vdash v : int$, then v is an integer.*

Proof: by inspection on expression typing rules, heap value rules and subtyping inversion Lemma 41. □

Theorem 57 *If $freetvs(\Sigma) = \emptyset$, and $\Theta \vdash H : \Sigma$, and $\Theta;\Sigma \vdash V : \Gamma$ and $\Theta;\bullet;\Sigma;\Gamma \vdash E : T$, then either E is a value, or E can evaluate one step, that is, $\exists H'$, V' and E' such that $(H;V;E) \mapsto (H';V';E')$.*

Proof: by induction on expression typing rules.

Case int, **Case label**, **Case tag**: all the expressions are values already.

Case var: $E = x$

By $\Theta;\Sigma \vdash V : \Gamma$, domain$(\Gamma)$ = domain(V). From $x \in$ domain(Γ), we know $x \in$ domain(V). Let $H' = H$, $V' = V$ and $E' = V(x)$. By **ev_var** $(H;V;E) \mapsto (H';V';E')$.

Case error: $E = \mathrm{error}[\tau]$: by **ev_error**, E steps to itself.

Case object: $E = C(e)$ with subderivation $\Theta;\bullet;\Sigma;\Gamma \vdash e : R(C)$.

By induction hypothesis, either e is a value or $\exists H'$, V' and e' such that $(H;V;e) \mapsto (H';V';e')$. If e is a value, then E is a value. Otherwise, let $E' = C(e')$. By the congruence rule, $(H;V;E) \mapsto (H';V';E')$.

Case c2r_c: $E = \mathrm{c2r}(e)$ with subderivation $\Theta;\bullet;\Sigma;\Gamma \vdash e : C$.

By induction hypothesis, either e is a value or $\exists H'$, V' and e' such that $(H;V;e) \mapsto (H';V';e')$. If e is a value, then by Lemma 56 $e = C(v)$. Let $E' = v$. By **ev_c2r** $(H;V;E) \mapsto (H';V';E')$. Otherwise let $E' = \mathrm{c2r}(e')$. By the congruence rule $(H;V;E) \mapsto (H';V';E')$.

Case c2r_tv: not applicable because by Lemma 17, the subderivation $\Theta;\bullet;\Sigma;\Gamma \vdash e : \alpha$ is invalid.

Case record: $E = \mathrm{new}[\tau]\{l_1 = e_1,\ldots,l_n = e_n\}$ with subderivations $\Theta;\bullet;\Sigma;\Gamma \vdash e_i : \tau_i\ \forall 1 \le i \le n$.

By induction hypothesis, each subexpression e_i either is a value or can evaluate one more step.

If all e_i are values, then let $H' = H, \ell \leadsto \{l_1 = e_1,\ldots,l_n = e_n\}$ (ℓ is a fresh label), $V' = V$ and $E' = \ell$. By **ev_record** $(H;V;E) \mapsto (H';V';E')$.

If $\exists e_i$ such that $e_1,\ldots,e_{i-1}$ are values and $\exists H',V',e_i'$ such that $(H;V;e_i) \mapsto (H';V';e_i')$ … $\mathrm{new}[\tau]\{l_1 = e_1,\ldots,l_i = e_i',\ldots,l_n = e_n\}$. By the congruence rule, $(H;V;E) \mapsto (H';V';E')$.

Case field: $E = e.l_i$ with subderivation $\Theta;\bullet;\Sigma;\Gamma \vdash e : \{l_1^{\phi_1} : \tau_1,\ldots,l_n^{\phi_n} : \tau_n\}$ and $1 \le i \le n$

By induction hypothesis, either e is a value or e can evaluate one step.

If e is a value, by canonical form Lemma 56, e is a label and $H(e) = \{l_1 = v_1,\ldots,l_n = \ldots$ $H' = H$, $V' = V$ and $E' = v_i$. By **ev_field** $(H;V;E) \mapsto (H';V';E')$.

If $\exists H',V',e'$ such that $(H;V;e) \mapsto (H';V';e')$, then let $E' = e'.l_i$ and by the congruence rule … $(H';V';E')$.

In the rest cases, we state only which evaluation rule to apply, but omit the new H', V' or …

Case assignR $E = e_1.l_i := e_2$ in e_3 with subderivations $\Theta;\bullet;\Sigma;\Gamma \vdash e_1 : \{l_1^{\phi_1} : \tau_1,\ldots,l_i^M \ldots$ $\}$ and $\Theta;\bullet;\Sigma;\Gamma \vdash e_2 : \tau_i$.

By induction hypothesis, either e_1 is a value or e_1 can evaluate one step. Similarly, either … e_2 can evaluate one step.

If both e_1 and e_2 are values, then by canonical form Lemma 56 e_1 is a label and $H(e_1) = \{l_1 = v_1,\ldots,l_n = \ldots$ $\ldots\}$, and by **ev_assignR** E can evaluate one step.

If either e_1 or e_2 can evaluate one step, by the congruence rule, E can evaluate one step.

Case array $E = \mathrm{new}[e_0,\ldots,e_{n-1}]^\tau$ with subderivations $\Theta;\bullet;\Sigma;\Gamma \vdash e_i : \tau\ \forall 0 \le i \le n-1$

If e_{t1} or e_{t2} can evaluate one step, then by the congruence rule E can evaluate one step.

Case ifTag_tv: $E = \mathrm{ifEqTag}^\tau(e_{t1}, e_{t2})$ then e_1 else e_2 with subderivation $\Theta;\bullet;\Sigma;\Gamma \vdash e_{t1} : \mathrm{Tag}(\gamma)$. Not applicable because the subderivation is invalid by Lemma 17.

Case sub: $E = e$ with subderivation $\Theta;\bullet;\Sigma;\Gamma \vdash e : \tau_1$.

By induction hypothesis, either e is a value or e can evaluate one step. That is, either E is a value or E can evaluate one step. □

Corollary 58 (progress) *If $\Sigma \vdash P : \tau$, then either the main expression in P is a value, or $\exists P'$ such that $P \mapsto P'$.*

Theorem 59 *LIL_C is Sound. Well-typed LIL_C programs do not get stuck.*

Proof: by progress and preservation. □

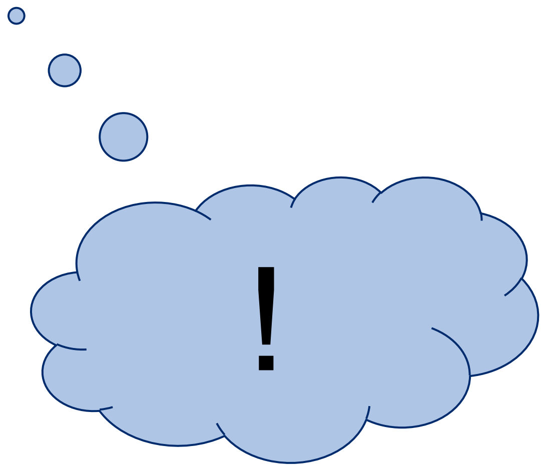
Theorem 48 *Type checking of LIL_C is decidable.*

Proof: To decide whether $\Theta;\Delta;\Sigma;\Gamma \vdash e : \tau$ holds, we can first get the minimal type τ_m of e such that $\Theta;\Delta;\Sigma;\Gamma \vdash e : \tau_m$, then test whether $\Theta;\Delta \vdash \tau_m \le \tau$. Because both minimal typing and subtyping are decidable, type checking is decidable. □

E Soundness

We’ve Got a Problem Here

- As a practical matter, a large fraction of the proofs being done about programming languages today are not possible for humans to check.



Are Proof Checkers the Answer?

- Proof checking (or "proof assistant") technology has made amazing strides in recent years
  - Several mature systems
    - E.g., Coq, HOL, Isabelle, Twelf, MetaPRL, etc., etc.
  - Some very impressive achievements in several domains

A Marriage Made in Heaven

So we've got…

A. A community with a burning need

B. A community with a great technology

Can we put them together?

Lots of Work Already Underway

- Leroy's verified C compiler
- Nipkow et al's formalization of a large part of Java
- Appel et al's Foundational Proof-Carrying Code project
- Crary et al's machine-checked development of a typed assembly language
- Harper et al's formalization of Standard ML
- Sewell et al's formalization of TCP/IP
- Etc., etc.

But We're Not There Yet!

Challenges:

- Current achievements are mostly heroic efforts by heroic individuals and small teams
- Many different proof assistants and diverse technical machinery
- Lots of black magic; high cost of entry; little sharing of knowledge across projects
- Some significant unresolved technical issues
  - Lightweight methods for reasoning about variable binding
  - Scalability and modularity of proofs

What's needed is some synergy…

Vision

A world where *every* PL paper is accompanied by a machine-checked appendix

- Plan:
  - Identify current best practices
  - Gather community consensus around them
  - Build additional tools and other infrastructure as needed
  - Address technical challenges specific to programming language research

First step…

The PoplMark Challenge

- A set of benchmark problems to help evaluate progress in the area
  - Based around metatheory of F<:, a typed lambda-calculus with polymorphism and subtyping
  - Presented at TPHOLs 2005
- Has generated tremendous interest in both PL and theorem proving communities
  - *many* solutions submitted
  - 6 different proof assistants
  - 7 different treatments of binding
- Much has been learned
  - JAR special issue CFP now open

Community Development

- Wiki & Mailing list for POPLmark challenge
  - Gathering place for news/solutions/discussion/advice
- Workshop on Mechanizing Metatheory
  - 4th Wksp, 4 Sep 2009, Edinburgh
- *Using Proof Assistants for Programming Language Research*
  or, How to write your next POPL paper in Coq
  - Tutorial for novices
  - POPL tutorial, January, 2008
  - Oregon Summer School, June 2008

Engineering Formal Metatheory

Resolving technical issues

- *Engineering Formal Metatheory*
  - Brian Aydemir, Arthur Charguéraud, Benjamin Pierce, Randy Pollack, and Stephanie Weirich
  - POPL 2008
- Describes a lightweight first-order methodology for representing binding and specifying induction principles
- Two essential components:
  - Locally nameless representation
  - Cofinite name quantification

What is so hard about binding?

- Alpha-equivalence
  - Identify "λx.x" and "λy.y"
- Barendregt convention
  - Assumption that bound variables are "sufficiently fresh"

```
Inductive exp : Set :=
 | var : atom -> exp
 | abs : atom -> exp -> exp
 | app : exp -> exp -> exp.
```

Alpha-equivalence

- Important when we need to compare terms with binding structure:
  - Type system of polymorphic language
  - Confluence for pure lambda calculus

- Formalism simpler if alpha-equals is "="

```
Lemma preservation: forall G e t,
  typing G e t -> red e e' ->
    exists t', alpha_eq t t' /\
    typing G e' t'.

Lemma preservation: forall G e t,
   typing G e t -> red e e' -> typing G e' t.
```

Barendregt Convention

```
Fixpoint subst (x:atom) (u:exp) (e:exp)
  {struct e} :=
match e with
 | var y      => if x == y then u else e
 | app t1 t2 => app (subst x u t1)
                    (subst x u t2)
 | abs z t    => abs z (subst x u t)
end.
```

- What if z == x?
- What if z in free variables of u?

Existing approaches

- No completely satisfactory mechanism for binding
  - **Name representation**: must explicitly rename terms, define alpha-equivalence
  - **de Bruijn indices**: difficult to work with as must shift indices
  - **Nominal logic**: only available in Isabelle/HOL
  - **HOAS**: exotic terms, specialized logic

Locally nameless representation

Names for free variables

de Bruijn indices for bound variables

Example: Untyped lambda calculus terms

$$t ::= \mathrm{bvar}\ i \mid \mathrm{fvar}\ x \mid \mathrm{app}\ t_1\ t_2 \mid \mathrm{abs}\ t$$

$\lambda x.\ \lambda y.\ (x\ y)\ z$ and $\lambda w.\ \lambda v.\ (w\ v)\ z$

represented as

abs (abs (app (app (bvar 1) (bvar 0)) (fvar "z"))

```
Inductive exp : Set :=
 | bvar : nat -> exp
 | fvar : atom -> exp
 | abs  : exp -> exp
 | app  : exp -> exp -> exp.
```

Basic operations

- Variable opening

  t^u replace bound index 0 with exp u

- Free variable calculation

  $\mathrm{FV}\ t$ finite set of free atoms in t

- Substitution

  $[x \mapsto u]t$ replace free variable x with u

- All operations have *simple* definitions.

Variable opening

```
Fixpoint open_rec (k : nat) (f : exp)
 (e : exp)  {struct e} : exp :=
  match e with
  | bvar i    => if k === i then f else e
  | fvar x    => fvar x
  | abs e1  =>
     abs (open_rec (1 + k) f e1)
  | app e1 e2 => app (open_rec k f e1)
                     (open_rec k f e2)
end.

Notation "e ^ u" := (open_rec 0 u e).
```

Free variable substitution

```
Fixpoint subst (z : atom) (u : exp)
  (e : exp) {struct e} : exp :=
  match e with
  | bvar i    => bvar i
  | fvar x    => if x == z then u else e
  | abs T e1  => abs T (subst z u e1)
  | app e1 e2 => app
      (subst z u e1) (subst z u e2)
end.
```

Free variable calculation

```
Fixpoint fv (e : exp)
 {struct e} : atoms :=
  match e with
   | bvar i   => {}
   | fvar x   => singleton x
   | abs e1   => (fv e1)
   | e1 e2    => (fv e1)
                  `union` (fv e2)
```

Local closure

- Not all members of type term are lambda calculus terms
  - abs (bvar 3)?
- Predicate lc picks out members datatype that are *locally-closed*
- Definitions respect local closure
  - *If* $lc\ u$ *and* $lc\ t$, *then* $lc\ ([x \mapsto u]t)$
  - *If* $t \rightarrow t'$ *then* $lc\ t$ *and* $lc\ t'$

Managing local closure

- Many theorems need not refer to local closure
  - *If* $t \to t'$ *and* $t \to t''$, *then* $t' = t''$
- Some theorems require it, tactics discharge assumptions
  - *If* $x \neq y$ *and* $lc\ u$, *then* $[x \mapsto u](t^{y}) = ([x \mapsto u]t)^{y}$

Induction principles

Definition of typing rules generates an induction principle for typing derivations

$$\frac{ok\ E \qquad (x{:}T) \in E}{E \vdash \mathrm{fvar}\ x : T}$$

$$\frac{E \vdash t_1 : S \to T \qquad E \vdash t_2 : S}{E \vdash \mathrm{app}\ t_1\ t_2 : T}$$

$$\frac{x \notin \mathrm{FV}\ t \cup dom(E) \qquad E,x{:}S \vdash t^x : T}{E \vdash \mathrm{abs}\ t : S \to T}$$

Exists vs. Cofinite Quantification

Induction hypothesis holds for some particular, unknown x

$$\frac{x \notin \mathrm{FV}\ t \cup dom(E) \qquad E,x{:}S \vdash t^x : T}{E \vdash \mathrm{abs}\ t : S \to T}$$

Induction hypothesis holds for all but some finite set of variables.

$$\frac{\forall x \notin L \qquad E,x{:}S \vdash t^x : T}{E \vdash \mathrm{abs}\ t : S \to T}$$

Weakening Lemma

If $E, G \vdash t : T$ and $ok\ (E\ ,\ F\ ,\ G)$
then $E, F, G \vdash t : T$

Proof (?):

by induction on $E,G \vdash t : T$

Abstraction case:

$$\frac{x \notin \mathrm{FV}\ t \cup dom(E,G) \qquad E,G,x{:}S \vdash t^x : T}{E,G \vdash \text{abs}\ t : S \to T}$$

WTP $E,F,G \vdash \text{abs}\ t : S \to T$

By typing rule, holds if $E,F,G,x{:}S \vdash t^x : T$
and $x \notin \mathrm{FV}\ t \cup dom(E,F,G)$

Weakening Lemma

If $E, G \vdash t : T$ and $ok\ (E\ ,\ F\ ,\ G)$
then $E, F, G \vdash t : T$

Proof: by induction on $E,G \vdash t : T$

abstraction case: $$\frac{\forall x \notin L \quad E,G,x{:}S \vdash t^x : T}{E,G \vdash \text{abs } t : S \to T}$$

WTP: $E,F,G \vdash \text{abs } t : S \to T$

By rule, holds if exists a set L', such that
$\forall x \notin L',\ \ E,F,G,x{:}S \vdash t^x : T$

IH: $\forall x \notin L,\ ok\ (E,F,G,\ x{:}S) \Rightarrow E,F,G,x{:}S \vdash t^x : T$

Choose $L' = dom\ (E,F,G) \cup L$

By definition, $\forall x \notin L',\ \ ok\ (E,F,G,x{:}S)$

Renaming lemma

- Similar reasoning proves substitution lemma
  If $E, x{:}S, F \vdash t : T$ and $E \vdash u : S$
  then $E,F \vdash [z \mapsto u]t : T$

- Important corollary of substitution and weakening

  Renaming:

  If $x \notin \mathrm{FV}\ t \cup dom\ E$ and $E, x{:}S \vdash t^x : T$
  then for all $y \notin \mathrm{FV}\ t \cup dom\ E,\quad E, y{:}S \vdash t^y{:}T$

Corollaries of renaming

- Renaming lemma gives strongest intro form:

  If exists x, s.t. $x \notin \mathrm{FV}\ t \cup dom\ E$ and
  $E, x{:}S \vdash t^x : T$ then $E \vdash \mathrm{abs}\ t : S \rightarrow T$

- And strongest inversion principle:

  If $E \vdash \mathrm{abs}\ t{:}\ S \rightarrow T$ then

  for all $x \notin \mathrm{FV}\ t \cup dom\ E,\ \ E, x : S \vdash t^x : T$

Equivalence of systems

- Renaming lemma also shows equivalence of two systems:

$E \vdash t : T$ with exists-fresh rule for abs
if and only if
$E \vdash t : T$ with cofinite rule for abs

- We are proving properties about the language we actually care about!

General Form of Development

- Def. of language syntax
- *Def. of variable opening and local closure*
- Def. of free variable substitution
- Def. of free variable function
- *Interaction lemmas*
- Def. of semantic relations *using cofinite quantification for binders*
- *Show semantic relations respect local closure*
- Substitution and weakening lemmas for each relation w/ binding
- Preservation and Progress
- *Derive renaming lemmas*

Code Distribution

- Reference *examples* & supporting experience
  - multiple calculi (STLC, F<:, CoC)
  - multiple theorems (type soundness, confluence)
- A *library* that supports this methodology
  - atoms, finite sets
  - reasoning about freshness
  - representing environments

Let's look closer at POPLmark…

Fsub_Definitions.v

syntax of types and terms, defn. of typing relation…

"The TCB"

Fsub_Lemmas.v

facts about well-formedness, environments…

"Things we should have proved on paper but didn't"

Fsub_Infrastructure.v

facts about syntax, substitution…

"The overhead"

Fsub_Soundness.v

Substitution lemma, preservation, progress…

"The meat"

Fsub_Definitions.v

syntax

open

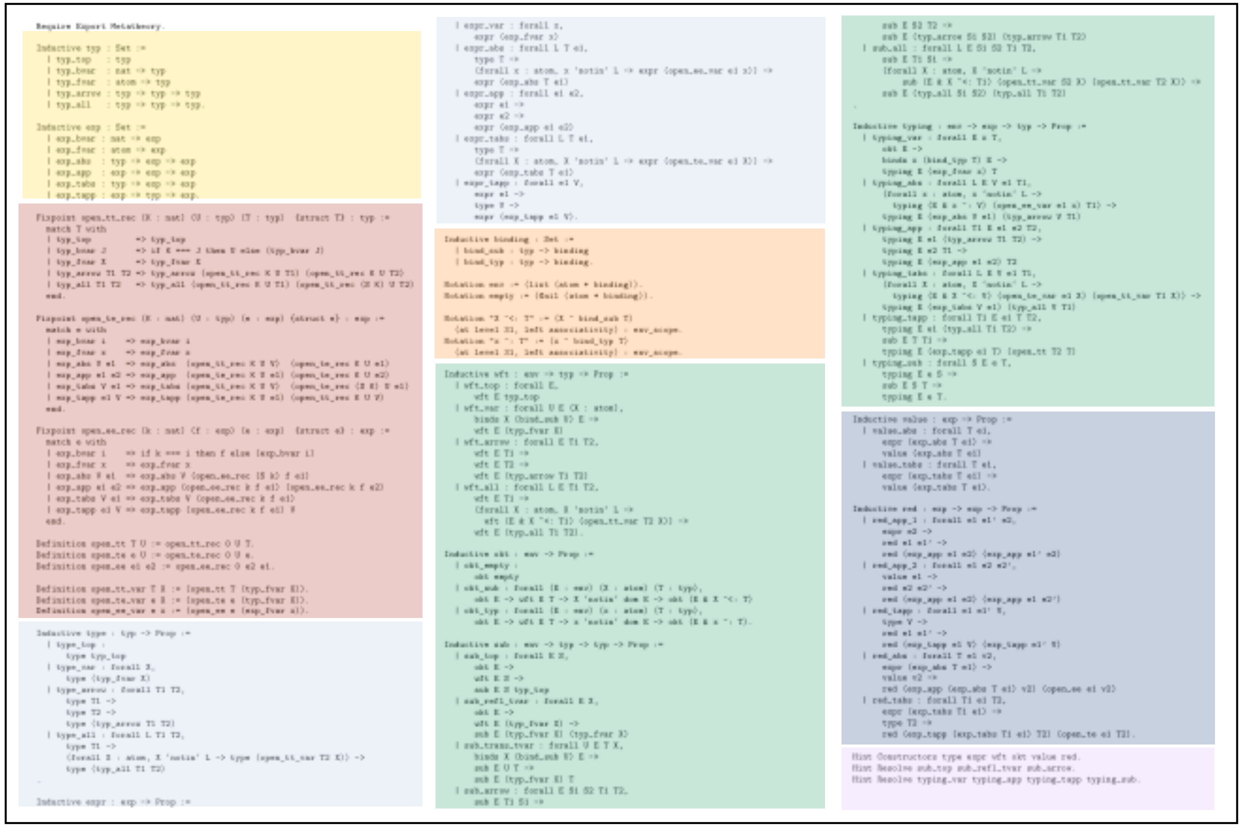
local closure

environments

typing & subtyping

operational semantics

automation hints



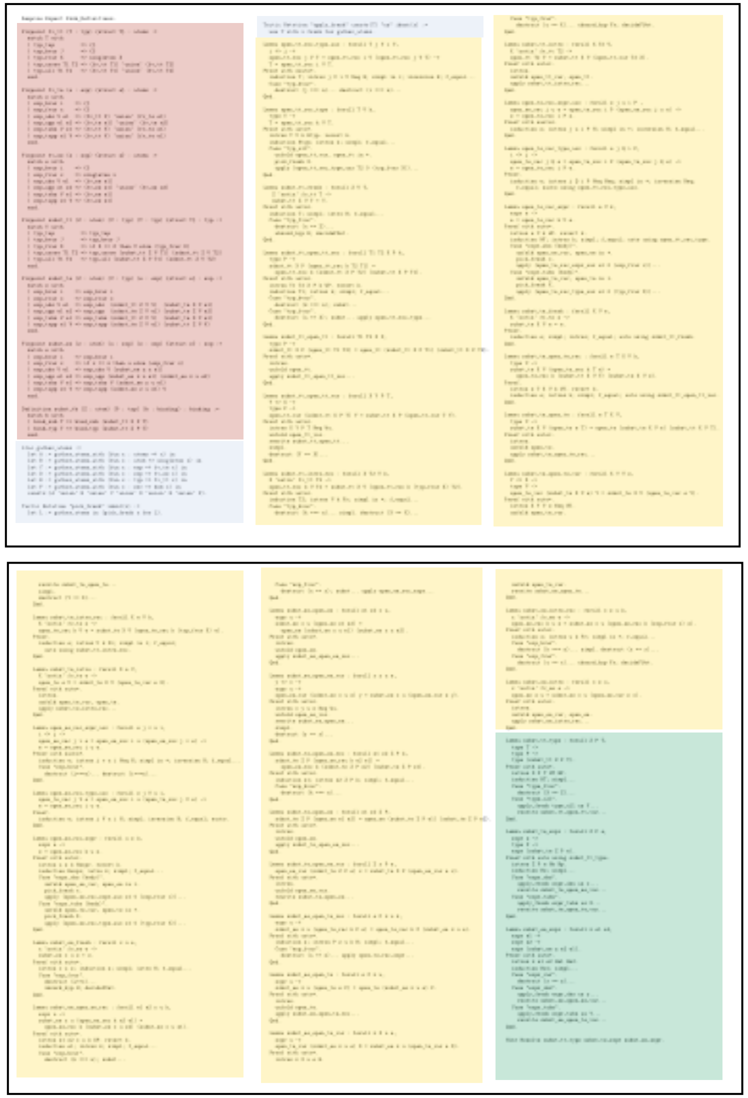
Fsub_Infrastructure.v

Basic operations (fv and subst)

Tactic to choose fresh atoms

Properties of open, fv and subst

Properties of subst and local closure



Fsub_Lemmas.v

type well-formedness

environment well-formedness

substitution in environments

regularity of relations

hints about local closure

Fsub_Soundness.v

Proofs of lemmas from the Appendix of POPLmark paper

Comparison POPLmark 1A

Author	Binding	Lemmas	Proof steps
Vouillon	de Bruijn	30	402
Leroy	Locally nameless	49	495
Stump	Levels/names	56	938
Hirschowitz & Maggesi	de Bruijn (nested datatype)	49	1574
Chlipala	Locally nameless	23	75
This work	Locally nameless	22	101

McKinna & Pollack Rule

Exists-fresh: IH holds for some fresh x

$$\frac{x \notin \mathrm{FV}\ t \cup dom(E) \qquad E, x{:}S \vdash t^x : T}{E \vdash \mathrm{abs}\ t : S \to T}$$

Cofinite: IH holds for all but some unknown set

$$\frac{\forall x \notin L \qquad E, x{:}S \vdash t^x : T}{E \vdash \mathrm{abs}\ t : S \to T}$$

Forall-fresh: IH holds for all fresh variables

$$\frac{\forall x \notin \mathrm{FV}\ t \cup dom(E) \qquad E, x{:}S \vdash t^x : T}{E \vdash \mathrm{abs}\ t : S \to T}$$

Forall vs. Cofinite

Define system with forall-fresh rules

Define swapping and show relations are stable under swapping

Show exists-fresh intro

Prove weakening and substitution

Prove type soundness

Define system with cofinite rules

Prove weakening and substitution

Show exists-fresh intro

Prove type soundness

Conclusions

- Can use Coq's standard mechanisms for reasoning (inductive defs, tactics, etc.)
- Swapping does not appear to be essential.
- Seldom need to rename during proofs. IH applies to an infinite # of suitably fresh variables.
- Specialized tactics help
  - local closure obligations
  - fresh variable introduction

Thanks to

Brian Aydemir

Arthur Charguéraud (INRIA)

Randy Pollack (Edinburgh)

Peter Sewell (Cambridge)

Aaron Bohannon

Matthew Fairbairn (Cambridge)

J. Nathan Foster

Benjamin Pierce

Jeffrey Vaughan

Dimitrios Vytiniotis

Geoffrey Washburn

Steve Zdancewic

Vouillon / de Bruijn indices

```
Fixpoint subst (t : term) (x : nat) (t' : term)
  {struct t} : term :=
  match t with
  | var y =>
      match lt_eq_lt_dec y x with
      | inleft (left _)  => var y
      | inleft (right _) => t'
      | inright _        => var (y - 1)
      end
  | abs T1 t2  =>
     abs T1 (subst t2 (1 + x) (shift 0 t'))
  | app t1 t2  => app (subst t1 x t')
     (subst t2 x t')
  | tabs T1 t2 => tabs T1
     (subst t2 x (shift_typ 0 t'))
  | tapp t1 T2 => tapp (subst t1 x t') T2
  end.
```