CTI-LIB: a Coq Library for PL Meta-Theory with Concrete Names

Aaron Stump

Computer Science and Engineering
Washington University
St. Louis, Missouri, USA

Contributions from Aayush Munjal, Michael Zeller.

Supported by NSF CCF-0448275.

CTI-LIB Goals

- "Contextual Term Interpretations Library".
- Support PL meta-theory in Coq with concrete names.
- Provide generic datatype for terms with binders.
- Provide recursion/induction principles for such terms.
- Define operations like substitution generically.
- Prove theorems like Substitution Lemma generically.
- Drive development by case studies.

Concrete Names vs. de Bruijn Indices

- Pros for concrete names:
  - Languages typically defined using named variables.
  - Tools support named variables.
  - There is a gap if meta-theory done with de Bruijn indices.
  - De Bruijn indices can be non-intuitive, tedious to work with.
- Cons for concrete names:
  Capture-avoiding substitution not easy to define.

Rest of Talk

- Generic Coq datatype of terms with binders.
- Defining functions by contextual term interpretation (CTI).
- An induction principle for CTIs.
- Alpha-canonical form and substitution.
- Use case study for examples:
  - Type preservation for a simply typed λ-calculus.
  - 2 abstractors: CBV λ_f and “transparent” λ_t.
  - Evaluation under λ_t is an additional challenge.

The `trm` Datatype

- Terms are uses of named variables or applications of operators.
- Names specified by a `NAMES` module:
  - A type `name`.
  - Computable isomorphism from `name` to the natural numbers.
- Operators specified in a `SIG` module:
  - A type op for operators, with decidable equality.
  - Arity functions: for each op, how many
    - Bound variables
    - “Non-governed” subterms
    - “Governed” subterms
  - Annotation type function: for each op:
    - a Coq `Set` for annotations
    - decidable equality on those annotations
- Dependent types ensure correct numbers of subterms.

The Coq Definition of `trm`

```
Module TRM(s:SIG)(n:NAMES).

Export s.
Export n.

Inductive trm : Set :=
  var : name -> trm
| exp : forall o:op,
        anno o ->
        trms (ar_ng o) -> (* not governed *)
        llist name (ar_b o) -> (* bound variables *)
        trms (ar_gv o) -> (* governed *)
        trm
with trms : nat -> Set :=
  trmsn : trms 0
| trmsc : forall n:nat, trm -> trms n -> trms (S n).

...
```

Example: Simply-Typed Lambda Terms

```
Module ST_SIG <: SIG.
  Inductive _op : Set :=
      _arrow : _op
    | _base : btp -> _op.
  ...
End ST_SIG.
Module ST := TRM ST_SIG NAT_NAMES.
Definition tp := ST.trm.

Module LAM_SIG <: SIG.
  Inductive _op : Set :=
      _lam : bool -> _op
    | _app : _op.
  Definition anno := fun o:op =>
   match o with
     _lam _ => tp
    | _app => unit
   end.
...
```

Contextual Term Interpretation

- Define function from `trm` to A by interpretation.
- So $[\![t]\!] : A$.
- User provides interpretations of operators.
- Library implements homomorphic extension to terms.
- To handle variables, the interpretation uses a context:
  - $\Gamma[\![t]\!] : A$.
  - Γ is a list of pairs of names and elements of A.
  - Interpret free variables as their values in Γ.
  - User provides function for free variables not declared in Γ.
- Interpretation of (binding) operator shows how to grow Γ:

$$\Gamma[\![f\ d\ \bar{n}\ \bar{x}\ \bar{g}]\!] = [\![f]\!]\ d\ (\Gamma[\![\bar{n}]\!])\ (\lambda \bar{a}.(\Gamma, \bar{x} \mapsto \bar{a})[\![\bar{g}]\!])$$

Example: Computing Free Variables

Interpret generic terms into *list name*.

$$
\begin{array}{rcl}
[\![f]\!] & := & \lambda d.\lambda \bar{N}.\lambda B. \\
 & & \quad (\cup \bar{N}) \ \cup \ (\cup (B \ \bar{nil})) \\
[\![x]\!] & := & [x] \text{ (for undeclared variables x)}
\end{array}
$$

Example: Computation of Simple Type

Interpret lambda terms into *option tp*.

$$
\begin{array}{rcl}
[\![lam\ b]\!] & := & \lambda T.\lambda N.\lambda B. \\
 & & \quad do\ R \leftarrow B\ (Some\ T) \\
 & & \qquad (Some\ (arrow\ T\ R)) \\
[\![app]\!] & := & \lambda\_.\lambda N.\lambda B. \\
 & & \quad do\ T_0 \leftarrow N_0 \\
 & & \qquad T_1 \leftarrow N_1 \\
 & & \qquad if\ (T_0 = arrow\ T_1\ R) \\
 & & \qquad then\ (Some\ R) \\
 & & \qquad else\ None \\
[\![x]\!] & := & None\ \text{(for undeclared variables x)}
\end{array}
$$

CTIs in Coq

```
Definition interp_fv_t(A:Type) := name -> A.

Definition interp_exp_t(A:Type) :=
  forall o:op,
  anno o ->
  illist A (ar_ng o) ->
  (illist A (ar_b o) -> illist A (ar_gv o)) ->
  A.

Module Type CTI_SIG.
  Parameter A:Type.
  Parameter interp_fv : interp_fv_t A.
  Parameter interp_exp : interp_exp_t A.
End CTI_SIG.

Module CTI (u:CTI_SIG).
  Fixpoint interp(G : ctxt u.A)(t : trm)
             {struct t} : u.A := ...
```

An Induction Principle

For a CTI into A:

For a predicate P : $ctxt\ A \rightarrow trm \rightarrow A \rightarrow Prop$:

To prove $\forall\ (t : trm)\ (G : ctxt\ A), P\ G\ t\ (G[\![t]\!])$, it suffices to prove:

- $P\ G\ x\ (G[\![x]\!])$, when $x \in dom(G)$
- $P\ G\ x\ (G[\![x]\!])$, when $x \notin dom(G)$
- P is preserved from immediate subterms to terms, for any extension of the context.

Alpha-Canonization

- Put generic terms t into α-canonical form from d:

  Consecutive bindings on paths from the root of t bind consecutive variables, starting from the d'th.

- To prevent capture: $d > i, \forall x_i \in FV(t)$.
- Implemented as a CTI into $nat \rightarrow trm$.
- So $acanon\ G\ t\ d\ :\ trm$.
- Substitution can be carried out during α-canonization:

$$[M/x]_d N \quad := \quad acanon\ (\cdot, x \mapsto M)\ N\ d$$

CTI Substitution Lemma

Theorem

Let M and N be generic terms, and x a name.
Assume $d > i, \forall x_i \in FV(M)$.
Assume $d > i, \forall x_i \in (FV(N) \setminus \{x\})$.
For any CTI with domain A, and any A-context Γ,
For any equivalence relation $=_A$ on A, we have

$$\Gamma[\![[M/x]_d N]\!] =_A (\Gamma, (x \mapsto \Gamma[\![M]\!]))[\![N]\!]$$

- Proof by CTI induction (230 lines).
- Stronger induction hypothesis required.
- Proof relies on weakening by a context (275 lines).
- (Weakening, contraction, permutation proved for all CTIs).

Simple Type Preservation

- A small-step evaluation function defined as a CTI:
  - Interpret into (*bool* $*$ *nat*) $\rightarrow$ *trm*.
  - The bool tells whether or not to reduce β-redexes.
  - Results are α-canonical from the given *nat*.
- Computation of simple type ("CST") defined as a CTI.
- For type preservation:
  - Prove that evaluation preserves bound on free variables (250 lines).
  - Need CTI substitution lemma, specialized to FV.
  - Type preservation proof by CTI induction on CST (225 lines).
  - Need CTI substitution lemma, specialized to CST.
- Overall development for simple types: 900 lines.

Lessons and Issues

- Getting the right definitions astoundingly hard.
  - Exact definition of CTI.
  - Exact form of substitution lemma.
  - Still have some clutter: context invariants.
- Mixing internal and external verification is helpful:
  - Dependent type of terms removes need for *option*.
  - No lemmas about when we get *Some*.
  - Programming with dependent types is tricky.
  - Streicher's axiom K needed.
- Small set of concepts helps develop a more complete theory.
- Subtyping not definable by CTI (not recursive in a single term.)
- An issue with the Coq module system?
  - Datatype definitions are generative.
  - Modular development must be linearized.

Conclusion and Future Work

- CTI-LIB: PL meta-theory in Coq with concrete names.
- Generic datatype of terms.
- Central idea: contextual term interpretation.
- Generic lemmas available for any function defined by CTI.
- CTI substitution lemma based on alpha-canonical form.
- Current development around 6kloc Coq.
- Some clean-up required and documenting paper, then release.
- Further case studies to drive development.