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.
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 \( \lambda \)-calculus.
  ▶ 2 abstractors: CBV \( \lambda_f \) and “transparent” \( \lambda_t \).
  ▶ Evaluation under \( \lambda_t \) is an additional challenge.
The `tm` 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 $\text{trm}$

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.
...

Aaron Stump
CTI-LIB
Define function from \( \text{term} \) to \( A \) by interpretation.

So \( \llbracket t \rrbracket : A \).

User provides interpretations of operators.

Library implements homomorphic extension to terms.

To handle variables, the interpretation uses a context:

- \( \Gamma[\llbracket t \rrbracket] : A \).
  - \( \Gamma \) is a list of pairs of names and elements of \( A \).
  - Interpret free variables as their values in \( \Gamma \).
  - User provides function for free variables not declared in \( \Gamma \).

Interpretation of (binding) operator shows how to grow \( \Gamma \):

\[
\Gamma[\llbracket f \ d \ \bar{n} \ \bar{x} \ \bar{g} \rrbracket] = \llbracket f \rrbracket \ d \ (\Gamma[\llbracket \bar{n} \rrbracket]) \ (\lambda \bar{a}.(\Gamma, \bar{x} \mapsto \bar{a})[\llbracket \bar{g} \rrbracket])
\]
Example: Computing Free Variables

Interpret generic terms into \textit{list name}.

\[
\begin{align*}
[f] & := \lambda d.\lambda \tilde{N}.\lambda B. \\
                & (\cup \tilde{N}) \cup (\cup (B \tilde{nil})) \\
[x] & := [x] \text{ (for undeclared variables } x) 
\end{align*}
\]
Example: Computation of Simple Type

Interpret lambda terms into *option tp*.

\[
\begin{align*}
[lam \ b] & := \lambda \ T. \lambda \ N. \lambda \ B. \\
& \quad \text{do } R \leftarrow B \ (\text{Some } T) \\
& \quad \quad (\text{Some } (\text{arrow } T \ R))
\end{align*}
\]

\[
\begin{align*}
[app] & := \lambda _- \lambda \ N. \lambda \ B. \\
& \quad \text{do } T_0 \leftarrow N_0 \\
& \quad T_1 \leftarrow N_1 \\
& \quad \text{if } (T_0 = \text{arrow } T_1 \ R) \\
& \quad \quad \text{then } (\text{Some } R) \\
& \quad \quad \text{else } \text{None}
\end{align*}
\]

\[
[x] := \text{None} \ (\text{for undeclared variables } x)
\]
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 := ...

Aaron Stump
An Induction Principle

For a CTI into $A$:

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

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

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

- Put generic terms $t$ into $\alpha$-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 $nat \to trm$.
- So $acanon\ G\ t\ d:trm$.
- Substitution can be carried out during $\alpha$-canonization:

  $$[M/x]_d N := acanon(\cdot, x \mapsto M) N d$$
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 $\Gamma$,
For any equivalence relation $\equiv_A$ on $A$, we have

\[
\Gamma[[M/x]dN] \equiv_A (\Gamma, (x \mapsto \Gamma[M]))[N]
\]

- 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 \((\text{bool} \times \text{nat}) \rightarrow \text{trm}\).
  - The bool tells whether or not to reduce \(\beta\)-redexes.
  - Results are \(\alpha\)-canonical from the given \(\text{nat}\).

- 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.