Mechanized Reasoning for Binding Constructs in Typed Assembly Language Using Coq

Nadeem Abdul Hamid
Berry College, Mount Berry, GA
Overview

- Background
- Motivation
- Nature of TAL encoding
- What didn’t work
- What did work
- Conclusion
Motivation

- **Proof-carrying code** ("Syntactic approach")

HLL with type system

\[ \vdash P : \tau \]

Machine code with safety proof

\[ \text{safe}(M, SP) \]
Syntactic Approach: PCC

• Three pieces

\[ \forall P, \tau, M. (\vdash P : \tau \text{ and } P \Rightarrow M) \rightarrow \text{safe}(M, SP) \]

\[ \forall P, \tau, M. (\vdash P : \tau \text{ and } P \Rightarrow M) \]
\[ \rightarrow (\exists \tau', M'. \vdash \text{step}(P) : \tau' \text{ and } \text{step}(P) \Rightarrow M') \]

\[ P_0 : \tau_0 \text{ and } P_0 \Rightarrow M_0 \]
Need for Soundness Proof

\[ \forall P, \tau, M. (\vdash P : \tau \text{ and } P \Rightarrow M) \]
\[ \rightarrow (\exists \tau', M'. \vdash \text{step}(P) : \tau' \text{ and } \text{step}(P) \Rightarrow M') \]

- Given \( P \), need to know that \( \text{step}(P) \)
exists, and that \( \text{step}(P) : \tau' \)

(Standard ‘Progress’ and ‘Preservation’ lemmas of soundness proof)
Typed Assembly Language

- No term level variables
- Several prototypes:
  - Recursive types
  - Simple polymorphism
  - Polymorphism with regions, capabilities
TAL Example

(types) \[\tau ::= \alpha \mid \top \mid \text{int} \mid \forall \sigma\]

(code types) \[\sigma ::= \Gamma \mid [\alpha]\sigma\] (registers) \[r ::= r_0 \mid r_1 \mid \ldots \mid r_7\]

(register file type) \[\Gamma ::= \{r_0:\tau_0, \ldots, r_7:\tau_7\}\] (ints, addresses) \[i, f ::= 0 \mid 1 \mid 2 \mid \ldots\]

(type context) \[\Delta ::= \alpha_0, \alpha_1, \ldots, \alpha_k\] (word values) \[v ::= i \mid f \mid v[\tau]\]

(type list) \[\vec{\tau} ::= \tau_0, \tau_1, \ldots, \tau_k\] (register file) \[R ::= \{r_0 \mapsto v_0, \ldots, r_7 \mapsto v_7\}\]

(instructions) \[\nu ::= \text{add } r_d, r_s, r_t \mid \text{addi } r_d, r_s, i \mid \text{sub } r_d, r_s, r_t \mid \text{subi } r_d, r_s, i \mid \text{mov } r_d, r_s \mid \text{movi } r_d, i \mid \text{movf } r_d, f \mid \text{bgti } r_s, i, f[\vec{\tau}] \mid \text{tapp } r_d[\tau]\]

(instr sequences) \[I ::= \nu; I \mid \text{jd } f[\vec{\tau}] \mid \text{jmp } r\]

(code values) \[c ::= \text{code } \sigma. I\]

(code heap) \[C ::= \{f_0 \mapsto c_0, \ldots, f_k \mapsto c_k\}\]

(program) \[P ::= (C, R, I)\]
What didn’t work

- In Coq, of course, full HOAS
- Impredicative inductive definition
  (definitions go through, but can’t reason on it)

Inductive $\Omega : \text{Kind} :=
\begin{align*}
\text{snat} : \text{Nat} \to \Omega \\
\text{sbool} : \text{Bool} \to \Omega \\
\rightarrow : \Omega \to \Omega \to \Omega \\
\text{tup} : \text{Nat} \to (\text{Nat} \to \Omega) \to \Omega \\
\forall_{\text{Kind}} : \Pi k : \text{Kind}. (k \to \Omega) \to \Omega \\
\exists_{\text{Kind}} : \Pi k : \text{Kind}. (k \to \Omega) \to \Omega
\end{align*}

Shao, et al. Type System for Certified Binaries

- Didn’t want any axioms, so no weak HOAS
What did work

- Lazy hack...
- ‘Locally-nameless’ first order encoding
  - Closed terms use de Bruijn encoding
  - Free variables => metalevel variables
- Neat substitution definition  (thanks to Valery Trifonov)
Results

- No variable contexts, ‘var’ terms
- No reasoning on substitution itself
  - For either type soundness, or any PCC proofs
  - Working with proofs, generating terms messy
Example

\[ \tau ::= \alpha \mid T \mid \tau_1 \to \tau_2 \mid \forall \alpha . \tau \]

- Encode with two inductive definitions
  - One representing terms with free variables as de Bruijn indices
  - One with no explicit variables
Example: Syntax Encoding

Inductive type : Set :=
| top : type
| arrow : type -> type -> type
| bind : ttype 1 -> type.

\[ \tau := \alpha | T | \tau_1 \rightarrow \tau_2 | \forall \alpha. \tau \]

Inductive ttype : nat -> Set :=
| tvar : forall i, ttype (S i)
| tlift : forall i, ttype i -> ttype (S i)
| ttop : ttype 0
| tarrow : forall i, ttype i -> ttype i -> ttype i
| tbind : forall i, ttype (S i) -> ttype i.
Substitution

Fixpoint subst_aux (i:nat) (t:ttype i) {struct t}
  : forall j, i=(S j) -> ttype j -> ttype j :=
  match t in (ttype i)
    return (forall j, i=S j -> ttype j -> ttype j) with
    | tvar n    => fun j _ e => e
    | tlift n t' => fun j (D:S n=S j) _
                     => eq_rec n _ t' j (myeqaddS n j D)
    | ttop      => fun j (D:0=S j) _   => 0_S_set _ j D
    | tarrow n t1 t2 => fun j (D:n=S j) e
                        => tarrow j (subst_aux n t1 j D e)
                        (subst_aux n t2 j D e)
    | tbind n t' => fun j (D:n=S j) e
                     => tbind j (subst_aux (S n) t' (S j) (eq_S _ _ D)
                                (tlift j e))
  end.
Notes on Substitution

- Substitution only defined for outermost variable... it’s all we needed in practice
- Dependent parameter tracks number of free variables
  - Maybe not useful other than as an exercise
  - Would complicate any reasoning
Between Representations

Fixpoint unlift_aux i (t:ttype i) {struct t} : 0=i -> type :=
  match t in (ttype i) return (0=i -> type) with
    | tvar n    => fun D => O_S_set _ n D
    | tlift n _ => fun D => O_S_set _ n D
    | ttop      => fun _ => top
    | tarrow n t1 t2 => fun D => arrow (unlift_aux n t1 D) (unlift_aux n t2 D)
    | tbind n t' => fun D => bind (eq_rec n (fun n => ttype (S n)) t' 0 (sym_eq D))
  end.

Definition unlift : ttype 0 -> type
  := fun t => unlift_aux 0 t (refl_equal 0).

Fixpoint lift (t:type) : ttype 0 :=
  match t with
    | top => ttop
    | arrow t1 t2 => tarrow 0 (lift t1) (lift t2)
    | bind t' => tbind 0 t'
  end.
Top-level Substitution

Definition subst : ttype 1 -> type -> type :=
  fun t e => unlift (subst_aux _ t _ (refl_equal 1) (lift e)).
Typing Rules

\[
\Delta, \alpha \vdash e : \tau \\
\frac{\Delta, \alpha \vdash e : \tau}{\Delta \vdash \text{all } \alpha.e : \forall \alpha.\tau}
\]

\[
\Delta \vdash \text{all } \alpha.e : \forall \alpha.\tau \\
\frac{\Delta \vdash (\text{all } \alpha.e)[\tau'] : \tau[\tau'/\alpha]}{\Delta \vdash (\text{all } \alpha.e)[\tau'] : \tau[\tau'/\alpha]}
\]
Encoding Typing Rules

Inductive typeof : exp -> type -> Prop :=
| wf_all : forall (e:exp) (t:ttype 1),
  (forall a, typeof e (subst t a)) ->
  typeof (all e) (bind t)
| ...
| ...  
| wf_tapp : forall (e:exp) (t'::type) (t:ttype 1),
  typeof (all e) (bind t) ->
  typeof (tapp (all e) t') (subst t t')

in evaluation rules:
tapp (all e) t' ==\Rightarrow e

Ties together for Preservation lemma...
Notes: Typing Rules

- Locally-nameless does not eliminate environments from encoding, in general
- In TAL, because there are no term level variables, there is nothing in the rules like: \[ \Delta, x : \tau \vdash \ldots \]
- More complex type level would not be as clean? (e.g. substitution under binders)
More Complex TAL

(kinds) $\kappa ::= \text{Type} | \text{Rgn} | \text{Cap}$

(constructors) $c ::= \tau | g | A$

(types) $\tau ::= \alpha | \text{int} | g \text{ handle} | \langle \tau_1 \times \tau_2 \rangle \text{ at } g | \forall [\Delta](A, \Gamma) | \mu \alpha. \tau$

(regions) $g ::= \rho | \nu$

(capabilities) $A ::= \epsilon | \emptyset | \{g^1\} | \{g^+\} | A_1 \oplus A_2 | \overline{A}$

(con. contexts) $\Delta ::= \cdot | \Delta, \alpha : \kappa | \Delta, \epsilon \leq A$

(register file types) $\Gamma ::= \{r0 : \tau_0, \ldots, r7 : \tau_7\}$

(region types) $\Upsilon ::= \{l0 : \tau_0, \ldots, ln : \tau_n\}$

(memory types) $\Psi ::= \{\nu0 : \Upsilon_0, \ldots, \nu_n : \Upsilon_n\}$

Figure 5.1: RgnTAL syntax.
RgnTAL Term Level

(labels) \( l, f ::= 0 | 1 | \ldots \)

(user registers) \( r ::= r0 \ | \ r1 \ | \ \ldots \ | \ r7 \)

(word values) \( v ::= i \ | \ v.l \ | \ f \ | \ handle \ (v) \ | \ v[c] \ | \ fold \ v \ as \ \tau \)

(register file) \( R ::= \{ r0 \mapsto v_0, \ldots, r7 \mapsto v_7 \} \)

(data heap values) \( h ::= (v_1, v_2) \)

(heap region) \( H ::= \{ l_0 \mapsto h_0, \ldots, l_n \mapsto h_n \} \)

(data memory) \( D ::= \{ v_0 \mapsto H_0, \ldots, v_n \mapsto H_n \} \)

(instructions) \( \tau ::= \text{add} \ r_d, r_s, r_t \ | \ \text{addi} \ r_d, r_s, i \ | \ \text{sub} \ r_d, r_s, r_t \ | \ \text{subi} \ r_d, r_s, i \ | \ \text{mov} \ r_d, r_s \ | \ \text{movi} \ r_d, i \ | \ \text{movf} \ r_d, f \ | \ \text{ld} \ r_d, r_s(i) \ | \ \text{st} \ r_d(i), r_s \ | \ \text{bgt} \ r_s, r_t, f \ | \ \text{bgti} \ r_s, i, f \ | \ \text{tapp} \ r[c] \ | \ \text{fold} \ r[\tau] \ | \ \text{unfold} \ r \)

(instr. sequences) \( I ::= \tau; I \ | \ \text{jd} \ f \ | \ \text{jmp} \ r \)

(code heap values) \( h ::= \text{code} \ [\Delta](A, \Gamma).I \ | \ \text{stub} \ [\Delta](A, \Gamma).\emptyset \)

(code memory) \( C ::= \{ f_0 \mapsto h_0, \ldots, f_n \mapsto h_n \} \)

(program) \( P ::= (D, R, I) \)
Caveat

- No reasoning needed about substitution for proofs, but actually producing typing derivation requires equality reasoning
- Can’t mix encoding styles

Inductive type : Set :=
| tint    : type                 (* int *)
| thandle : rgn -> type          (* p handle *)
| tpair   : type -> type -> rgn -> type (* t1 x t2 at p *)
| tabsr   : (rgn -> type) -> type (* \ p:Rgn. t *)
| tabst   : (ttype 1) -> type    (* \ t:Type. t' *)
| ...
Conclusion

- Locally-nameless (independently discovered) provides ‘non-intrusive’ treatment of binding constructs
- Much boilerplate code
- Parameterized definition of de Bruijn terms fun but complicate reasoning if it were needed
Thank you!
nadeem@acm.org