Mechanized Reasoning for Binding Constructs in Typed Assembly Language Using Coq

Nadeem Abdul Hamid

Berry College, Mount Berry, GA

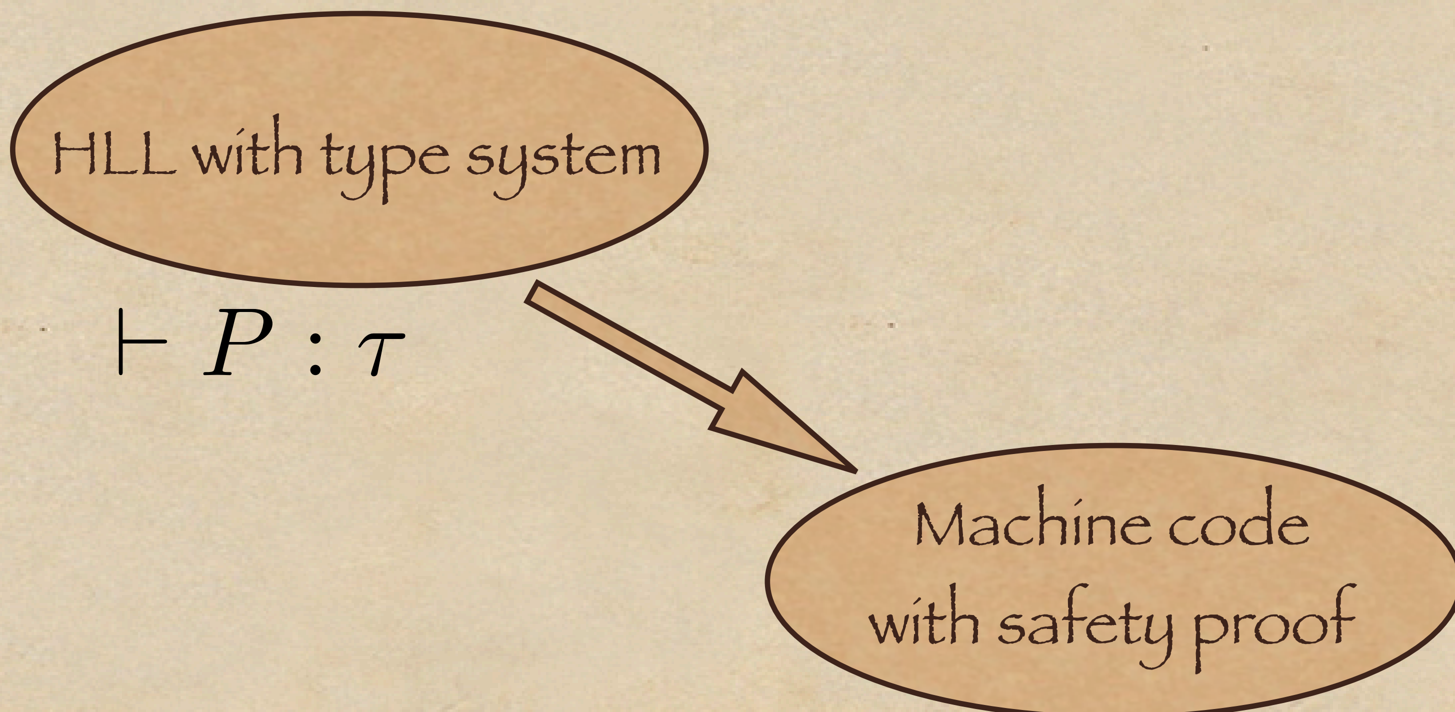
1st Informal ACM SIGPLAN Workshop on Mechanizing Metatheory (WMM 2006) --- September 21, 2006, Portland, OR

Overview

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

Motivation

- Proof-carrying code ("Syntactic approach")



$\vdash P : \tau$

$\text{safe}(M, \mathsf{SP})$

Syntactic Approach: PCC

- Three pieces

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

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

$$P_0 : \tau_0 \text{ and } P_0 \Rightarrow M_0$$

Need for Soundness Proof

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

- 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

$$
\begin{array}{llcl}
(\textit{types}) & \tau & ::= & \alpha \mid \top \mid \mathsf{int} \mid \forall\sigma \\
(\textit{code types}) & \sigma & ::= & \Gamma \mid [\alpha]\sigma \\
(\textit{register file type}) & \Gamma & ::= & \{\mathsf{r0}\!:\!\tau_0, \ldots, \mathsf{r7}\!:\!\tau_7\} \\
(\textit{type context}) & \Delta & ::= & \alpha_0, \alpha_1, \ldots, \alpha_k \\
(\textit{type list}) & \vec{\tau} & ::= & \tau_0, \tau_1, \ldots, \tau_k
\end{array}
$$

$$
\begin{array}{llcl}
(\textit{registers}) & \mathsf{r} & ::= & \mathsf{r0} \mid \mathsf{r1} \mid \ldots \mid \mathsf{r7} \\
(\textit{ints, addresses}) & i, f & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
(\textit{word values}) & v & ::= & i \mid f \mid v[\tau] \\
(\textit{register file}) & R & ::= & \{\mathsf{r0} \mapsto v_0, \ldots, \mathsf{r7} \mapsto v_7\}
\end{array}
$$

$$
\begin{array}{llcl}
(\textit{instructions}) & \iota & ::= & \mathsf{add}\ \mathsf{r}_d, \mathsf{r}_s, \mathsf{r}_t \mid \mathsf{addi}\ \mathsf{r}_d, \mathsf{r}_s, i \mid \mathsf{sub}\ \mathsf{r}_d, \mathsf{r}_s, \mathsf{r}_t \mid \mathsf{subi}\ \mathsf{r}_d, \mathsf{r}_s, i \\
& & & \mid \mathsf{mov}\ \mathsf{r}_d, \mathsf{r}_s \mid \mathsf{movi}\ \mathsf{r}_d, i \mid \mathsf{movf}\ \mathsf{r}_d, f \mid \mathsf{bgti}\ \mathsf{r}_s, i, f[\vec{\tau}] \mid \mathsf{tapp}\ \mathsf{r}_d[\tau] \\
(\textit{instr sequences}) & I & ::= & \iota; I \mid \mathsf{jd}\ f[\vec{\tau}] \mid \mathsf{jmp}\ \mathsf{r} \\
(\textit{code values}) & c & ::= & \mathsf{code}\ \sigma.\, I \\
(\textit{code heap}) & \mathcal{C} & ::= & \{f_0 \mapsto c_0, \ldots, f_k \mapsto c_k\} \\
(\textit{program}) & \mathcal{P} & ::= & (\mathcal{C}, R, I)
\end{array}
$$

What didn't work

- In Coq, of course, full HOAS
- Impredicative inductive definition

  (definitions go through, but can't reason on it)

$$
\begin{aligned}
\textsf{Inductive}\ \Omega\ :\ \textsf{Kind}\ :=\ & \textsf{snat} && : \textsf{Nat} \rightarrow \Omega \\
\mid\ & \textsf{sbool} && : \textsf{Bool} \rightarrow \Omega \\
\mid\ & \twoheadrightarrow && : \Omega \rightarrow \Omega \rightarrow \Omega \\
\mid\ & \textsf{tup} && : \textsf{Nat} \rightarrow (\textsf{Nat} \rightarrow \Omega) \rightarrow \Omega \\
\mid\ & \forall_{\textsf{Kind}} && : \Pi k : \textsf{Kind}.\ (k \rightarrow \Omega) \rightarrow \Omega \\
\mid\ & \exists_{\textsf{Kind}} && : \Pi k : \textsf{Kind}.\ (k \rightarrow \Omega) \rightarrow \Omega
\end{aligned}
$$

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

$$\tau := \alpha \mid \top \mid \tau_1 \to \tau_2 \mid \forall \alpha . \tau$$

- 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.
```

$$\tau := \ \alpha \mid \top \mid \tau_1 \to \tau_2 \mid \forall \alpha . \tau$$

```
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) _    => O_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
    | tbind n t' => fun D => bind (eq_rec n (fun n => ttype (S n)) t' 0 (sy
  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

$$\frac{\Delta, \alpha \vdash e : \tau}{\Delta \vdash \textsf{all } \alpha.e : \forall\alpha.\tau}$$

$$\frac{\Delta \vdash \textsf{all } \alpha.e : \forall\alpha.\tau}{\Delta \vdash (\textsf{all } \alpha.e)[\tau'] : \tau[\tau'/\alpha]}$$

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' ==> 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*)	κ	$::= \mathsf{Type} \mid \mathsf{Rgn} \mid \mathsf{Cap}$
(*constructors*)	c	$::= \tau \mid g \mid A$
(*types*)	τ	$::= \alpha \mid \mathsf{int} \mid g\ \mathsf{handle} \mid \langle \tau_1 \times \tau_2 \rangle\ \mathsf{at}\ g \mid \forall[\Delta](A, \Gamma) \mid \mu\alpha.\tau$
(*regions*)	g	$::= \rho \mid \nu$
(*capabilities*)	A	$::= \epsilon \mid \emptyset \mid \{g^1\} \mid \{g^+\} \mid A_1 \oplus A_2 \mid \overline{A}$
(*con. contexts*)	Δ	$::= \cdot \mid \Delta, \alpha : \kappa \mid \Delta, \epsilon \leq A$
(*register file types*)	Γ	$::= \{\mathsf{r0} : \tau_0, \ldots, \mathsf{r7} : \tau_7\}$
(*region types*)	Υ	$::= \{l_0 : \tau_0, \ldots, l_n : \tau_n\}$
(*memory types*)	Ψ	$::= \{\nu_0 : \Upsilon_0, \ldots, \nu_n : \Upsilon_n\}$

RgnTAL Term Level

(labels)	l, f	$::= \mathbf{0} \mid \mathbf{1} \mid \ldots$
(user registers)	r	$::= \mathsf{r0} \mid \mathsf{r1} \mid \ldots \mid \mathsf{r7}$
(word values)	v	$::= i \mid \nu.l \mid f \mid \mathsf{handle}\ (\nu) \mid v[c] \mid \mathsf{fold}\ v\ \mathsf{as}\ \tau$
(register file)	R	$::= \{\mathsf{r0} \mapsto v_0, \ldots, \mathsf{r7} \mapsto v_7\}$
(data heap values)	h	$::= (v_1, v_2)$
(heap region)	H	$::= \{l_0 \mapsto h_0, \ldots, l_n \mapsto h_n\}$
(data memory)	$\mathcal{D}$	$::= \{\nu_0 \mapsto H_0, \ldots, \nu_n \mapsto H_n\}$
(instructions)	ι	$::= \mathsf{add}\ r_d, r_s, r_t \mid \mathsf{addi}\ r_d, r_s, i \mid \mathsf{sub}\ r_d, r_s, r_t \mid \mathsf{subi}\ r_d, r_s, i$ $\mid \mathsf{mov}\ r_d, r_s \mid \mathsf{movi}\ r_d, i \mid \mathsf{movf}\ r_d, f \mid \mathsf{ld}\ r_d, r_s(i)$ $\mid \mathsf{st}\ r_d(i), r_s \mid \mathsf{bgt}\ r_s, r_t, f \mid \mathsf{bgti}\ r_s, i, f \mid \mathsf{tapp}\ r[c]$ $\mid \mathsf{fold}\ r[\tau] \mid \mathsf{unfold}\ r$
(instr. sequences)	I	$::= \iota; I \mid \mathsf{jd}\ f \mid \mathsf{jmp}\ r$
(code heap values)	$\overline{h}$	$::= \mathsf{code}\ [\Delta](A, \Gamma).I \mid \mathsf{stub}\ [\Delta](A, \Gamma).\emptyset$
(code memory)	$\mathcal{C}$	$::= \{f_0 \mapsto \overline{h}_0, \ldots, f_n \mapsto \overline{h}_n\}$
(program)	$\mathcal{P}$	$::= (\mathcal{D}, R, I)$

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