Coq Mechanization of Basic Core Fortress for Type Soundness

ERC Workshop (25 Aug to 28 Aug, 2010) Jieung Kim (KAIST) Sukyoung Ryu (KAIST)

Motivation

Fortress is a programming language for scientists and engineers and Basic Core Fortress (BCF) illustrates a basic core calculus for Fortress. Type soundness proof of BCF is shown in the Fortress language specification version 0.707 but mechanization of BCF is not yet done. We are working on mechanizing the BCF syntax and semantics and proving the type soundness of BCF with Coq, one of the practical proof assistant tool.

Basic Core Fortress

- Basic core calculus for Fortress
- All valid BCF programs are valid Fortress programs.
- The syntax of BCF allows only a small subset of the Fortress programming language to be formalized.
- BCF is quite similar to Featherweight Generic Java (FGJ).
  - Key differences : multiple inheritance and two kinds of classes (traits and objects)
- Syntax and some semantics of BCF

α, β			type variables
f			method name
x			field name
T			trait name
O			object name
τ, τ', τ''	::=	α	type
	|	σ	
σ	::=	N	named type
	|	$O[\![\overrightarrow{\tau}]\!]$	
N, M, L	::=	$T[\![\overrightarrow{\tau}]\!]$	non-leaf type
	|	Object	
e	::=	x	expression
	|	self	
	|	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
	|	$e.x$	
	|	$e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
fd	::=	$f[\![\alpha \texttt{ extends } \overrightarrow{N}]\!](\overrightarrow{x:\tau})\tau = e$	method definition
td	::=	$\texttt{trait}\ T[\![\alpha \texttt{ extends } \overrightarrow{N}]\!] \texttt{ extends } \{\overrightarrow{N}\}\ \overrightarrow{fd}\ \texttt{end}$	trait definition
od	::=	$\texttt{object}\ O[\![\alpha \texttt{ extends } \overrightarrow{N}]\!](\overrightarrow{x:\tau}) \texttt{ extends } \{\overrightarrow{N}\}\ \overrightarrow{fd}\ \texttt{end}$	object definition
d	::=	td	definition
	|	od	
p	::=	$\overrightarrow{d}\ e$	program

Evaluation rules: $p \vdash E[R] \longrightarrow E[e]$

[R-Field]
$$\frac{\texttt{object}\ O\ \_(\overrightarrow{x:\_})\ \_\ \texttt{end} \in p}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).x_i] \longrightarrow E[v_i]}$$

[R-Method]
$$\frac{\texttt{object}\ O\ \_(\overrightarrow{x:\_})\ \_\ \texttt{end} \in p \qquad mbody_p(f[\![\overrightarrow{\tau}]\!], O[\![\overrightarrow{\tau}]\!]) = \{(\overrightarrow{x}) \to e\}}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).f[\![\overrightarrow{\tau}]\!](\overrightarrow{v})] \longrightarrow E[[\overrightarrow{v}/\overrightarrow{x}][O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})/\texttt{self}][\overrightarrow{v}/\overrightarrow{x}]e]}$$

Method body lookup: $mbody_p(f[\![\overrightarrow{\tau}]\!], \tau) = \{(\overrightarrow{x}) \to e\}$

[Mb-Self]
$$\frac{\_\ C[\![\alpha \texttt{ extends } \_]\!]\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f[\![\alpha' \texttt{ extends } \_]\!](\overrightarrow{x:\_})\ \_ = e \in \{\overrightarrow{fd}\}}{mbody_p(f[\![\overrightarrow{\tau}]\!], C[\![\overrightarrow{\tau}]\!]) = \{[\overrightarrow{\tau}/\overrightarrow{\alpha}][\overrightarrow{\tau}/\overrightarrow{\alpha}](\overrightarrow{x}) \to e\}}$$

[Mb-Super]
$$\frac{\_\ C[\![\alpha \texttt{ extends } \_]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f \notin \{\overrightarrow{fname(fd)}\}}{mbody_p(f[\![\overrightarrow{\tau}]\!], C[\![\overrightarrow{\tau}]\!]) = \bigcup_{N_i \in \{\overrightarrow{N}\}} mbody_p(f[\![\overrightarrow{\tau}]\!], [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i)}$$

[Mb-Obj]
$$mbody_p(f[\![\overrightarrow{\tau}]\!], \text{Object}) = \emptyset$$

Steps to Implementation

1. **BCF--** ← **Current**
   - No multiple inheritance nor generic
2. **BCF-**
   - No generic
   → Need to express multiple inheritance with primitive recursion in Coq
3. **BCF**
   - Full features of BCF
   → Need to handle type substitution

Coq

Coq (http://coq.inria.fr/) is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs.

BCF-- vs Cast-Free FJ

- Cast-Free Featherweight Java (Cast-Free FJ): http://soft.vub.ac.be/~bdefrain/featherj/
- Trait and object definitions in BCF-- and class definitions in Cast-Free FJ
  - Two kinds of classes in BCF-- make it hard to use the metatheory library of Cast-Free FJ as it is.

get function in metatheory.v

```
Fixpoint get (x: atom) (E: list (atom * A)) : option A :=
  match E with
  | nil => None
  | (y,v)::E => if x == y then Some v else get x E
  end.
```

Cast-Free FJ

```
Notation ctable := (list (cname * (cname * flds * mths))).
```

→ Separate class tables as trait table and object table →

BCF--

```
Notation ttable := (list (tname * (tname * mths))).
Notation otable := (list (oname * (tname * flds * mths))).
```

- Advantages and disadvantages of separating trait table and object table
  - Advantages: easy to reuse the meatheory library of Cast-Free FJ, easy to prove the properties that are related to *field* declarations
  - Disadvantages: need to consider more cases, hard to prove some properties that are related to *method* declarations

```
Definition ob_extends (C : oname) (D : tname) : Prop :=
    exists fs, exists ms, (binds C (D, fs, ms) OT).

Definition tr_extends (C : tname) (D : tname) : Prop :=
    exists ms, (binds C (D, ms) TT).
```

```
Inductive sub_ty .........
  | s_ob_tapp : forall C D, ob_extends C D -> sub_ty (otyp2typ C) (ttyp2typ D)
  | s_tr_tapp : forall C D, tr_extends C D -> sub_ty (ttyp2typ C) (ttyp2typ D).
```

Current Implementation

Atom.v

Declares the `atom` type as the base type of every variable in our Coq implementation. The `atom` type describes structureless objects such that we can always generate one fresh from a finite collection. This file is taken from Cast-Free FJ.

```
Module Type ATOM.

  Parameter atom : Set.
  Parameter atom_fresh_for_list :
    forall (xs : list atom), {x : atom | ~ List.In x xs}.
  Parameter eq_atom_dec : forall x y : atom, {x = y} + {x <> y}.

End ATOM.
```

BCF--_Definition.v

Defines syntax and semantics definition of BCF--. This file includes:
- Syntax definition
- Some auxiliary functions to define semantics
- Dynamic semantics (evaluation context and evaluation rule)
- Static semantics (well-formed type, subtyping, and declaration typing)

```
Definition var := atom.
Definition fname := atom.
Definition mname := atom.
Definition oname := atom.
Definition tname := atom.
```
Use `atom` type for freshness.

```
Inductive exp : Set :=
    | e_var : var -> exp
    | e_self : var -> exp
    | e_new : oname -> list exp -> exp
    | e_field : exp -> fname -> exp
    | e_method : exp -> mname -> list exp -> exp.

Notation mths := (list (mname*(env*typ*exp))).
```

```
Inductive eval_rule : exp -> exp -> Prop :=
  | r_field : forall C fs es f e fes,
      ob_fields C fs ->
      env_zip fs es fes ->
      binds f e fes ->
      eval_rule (e_field (e_new C es) f) e
  | r_method : forall C m E t e es ves es0,
      ob_method C m (E, t, e) ->
      env_zip E es ves ->
      eval_rule (e_method (e_new C es0) m es) (subst_exp ((self, e_new C es0)::ves) e)
  ......
```

BCF--_Facts.v

Includes `Lemmas` and `Facts` to prove preservation and progress theorems. (Work in progress)

```
Scheme typing_typings_ind := Minimality for exp_ty Sort Prop
with wide_typing_typings_ind := Minimality for wide_typing Sort Prop
with wide_typings_typings_ind := Minimality for wide_typings Sort Prop.

Combined Scheme typings_mutind from typing_typings_ind, wide_typing_typings_ind, wide_typings_typings_ind.
```
For the case analysis of `exp_ty`, `wide_typing`, and `wide_typings` in the `progress'` theorem in BCF--_Properties.v

```
Module Type NoObj.
    Parameter ot_noobj: Object \notin dom OT.
    Parameter tt_noobj: Object \notin dom TT.
End NoObj.

Module NoObjFacts (Import H: NoObj).

    Fact obj_nofields : forall flds, tr_fields Object flds -> flds = nil.
    Proof.
      intros.
      inversion_clear H; subst; reflexivity.
    Qed.

.......
```
Properties that follow from the `ot_noobj` and `tt_noobj` hypotheses

```
Module Type OkTable.
    Parameter ok_ot: ok_otable OT.
    Parameter ok_tt: ok_ttable TT.
End OkTable.

Module OkTableFacts (Import H: OkTable).
.......
    Fact tr_method_implies_typing: forall t m E t0 e,
        tr_method t m (E,t0,e) ->
        exists2 t', sub_ty (ttyp2typ t) t' & wide_typing ((self,t'):::E) e t0.
......
```
Properties that follow from the `ok_ot` and `ok_tt` assumptions

AdditionalTatics.v

Includes some user-defined tactics to prove type soundness easily. This file is taken from Cast-Free FJ.

Metatheory.v

Provides a number of auxiliary constructs (and their properties) for the study of programming languages in Coq. Most parts in this library are variants of the list library. This file is taken from Cast-Free FJ.

BCF--_Properties.v

Proves type soundness of BCF--:
- Weakening (optional)
- Term substitution preserves typing
- Preservation Theorem
- Progress Theorem

```
Lemma weakening: forall E F G e t,
      ok (F++E++G) ->
      exp_ty (F++G) e t ->
      exp_ty (F++E++G) e t.
```
The `weakening` lemma is not used in the proof of the preservation theorem because Metatheory.v already handles it.

```
Lemma term_substitutivity: forall E E0 e t ds Eds,
      wide_typing E e t ->
      wide_typings E0 ds (imgs E) ->
      env_zip E ds Eds ->
      wide_typing E0 (subst_exp Eds e) t.
```
The `term_substitutivity` lemma proves that term substitution preserves typing, and it is used in the proof of the `preservation` theorem. (Check the red box below.) It is quite similar to the hand proof in the Fortress programming language specification version 0.707.

```
Theorem preservation: forall E e e' t,
      exp_ty E e t ->
      eval_rule e e' ->
      wide_typing E e' t.
  Proof.
    intros. generalize dependent t.
    induction H0; intros.
    Case "eval_field".
      inversion H2. inversion H6. subst.
      destruct H8 as (fs1, H8a, H8b).
      assert (fs0 = fs1) by (eapply ob_fields_fun; eassumption). subst.
      assert (fs1 = fs) by (eapply ob_fields_fun; eassumption). subst.
      destruct binds_zip with (1:=H15) (2:=H0) (3:=H8b) as (e0, Hb1, Hb2).
      assert (e0 = e) by (eapply binds_fun; eassumption). subst.
      exact Hb2.
    Case "eval_meth".
      inversion H1.
      SCase "eval_meth_with_trait_method".
        inversion H6.
      SCase "eval_meth_with_object_method".
        subst.
        assert (C0 = C) by (inversion H6; auto); subst.
        assert ((E0, t, e) = (ME, t0, body)) by (eapply ob_method_fun; eauto).
        subst injections.
        destruct ob_method_implies_typing with (1:=H8) as (H8a, t', H8b).
        eapply term_substitutivity; (try simpl; eauto).
    Case "eval_context".
      eapply preservation_over_ec; try eauto.
  Qed.
```

```
Theorem progress' :
    (forall E e t, exp_ty E e t ->
    E = nil -> val e \/ exists e', eval_rule e e')
    /\
    (forall E e t, wide_typing E e t ->
    E = nil -> val e \/ exists e', eval_rule e e')
    /\
    (forall E ds env, wide_typings E ds env ->
    E = nil -> (forall v, In v ds -> val v) \/
    exists2 EE, exps_context EE & (exists2 e0, (EE e0) =
    ds & (exists e0', eval_rule e0 e0'))).
Proof.
  apply typings_mutind; intros; subst E; specialize trivial.
  Case "t_var".
    contradiction (binds_nil H0).
  Case "t_self".
    contradiction (binds_nil H0).
  Case "t_new".
    destruct H2 as [H2 | (EE,H2a,(e0,H2b,(e0',H2c)))]; [ auto | ].
    subst es. eauto.
  Case "t_field".
    destruct H0 as [H0 | (e', H0)]; [ |right; eauto].
    destruct H0. inversion H. subst.
    destruct wide_typings_implies_zip with (1 := H8) as (Eds, Hzip).
    destruct H1 as ( fs0, H1a, H1b).
    assert (fs0 = fs) by (eapply ob_fields_fun; eassumption). subst.
    destruct binds_zip with (1:=H8) (2:=Hzip) (3:=H1b).
    eauto.
  Case "t_tr_meth".
    destruct H0 as [H0 | (e',H0)]; [ | right; eauto ].
    destruct H0. inversion H.
  Case "t_ob_meth".
    destruct H0 as [H0 | (e', H0)]; [ | right; eauto ].
    destruct H0. inversion H. subst.
    destruct wide_typings_implies_zip with (1:=H2) as (Eds,Hzip).
    eauto.
  Case "wt_sub". auto.
  Case "wts_nil". left. intros. contradiction H0.
  Case"wts_cons".
    destruct H0 as [H0 | (EE, H0a, (e0, H0b, H0c))].
    SCase "values es".
      destruct H2 as [H2 | (e', H2)].
      SSCase "value e".
        left. intros.
        destruct in_inv with (1:=H3); [ subst | ]; auto.
      SSCase "e_progress".
        right. exists (fun el => el::es); eauto.
    SCase "es_progress".
      subst es. right. exists (fun el => e::(EE el)); eauto.
Qed.
```
Has the same meaning with the `progress` theorem, but describes more statements to make it easy to prove each case.

Unlike in the hand proof, performs case analysis on the expression typing, which has more but simpler cases.

```
Theorem progress: forall e t,
    exp_ty nil e t ->
    val e \/ (exists e', eval_rule e e').
Proof.
  intros.
  destruct (progress') as (Hpro,_).
  apply Hpro with (1:=H). reflexivity.
Qed.
```
Prove it indirectly using the `progress'` theorem.