ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 NRF ERC

ROSAEC MEMO
ROSAEC-2011-011
May 24, 2011

Coq Mechanization of Featherweight Basic Core Fortress for Type Soundness

Jieung Kim, Sukyoung Ryu
KAIST
gbali@kaist.ac.kr, sryu@cs.kaist.ac.kr

May 24, 2011

1 Introduction

We verify the type soundness of Featherweight Basic Core Fortress (FBCF), a very small subset of the Fortress programming language [3] in Coq [2].

Fortress is a general-purpose, statically typed, component-based programming language designed for producing robust high-performance software with high programmability. The Fortress Language Specification Version 1.0 [5] presents the Basic Core Fortress (BCF) calculus, a basic core caculus for Fortress. We even more simplify BCF to FBCF to mechanize its type soundness proof in Coq.

The rest of this report is organized as follows. Section 2 presents a calculus of FBCF. From Section to Section 5, we present some Coq libraries that are taken from the POPLmark Wiki [4] and the Cast-Free Featherweight Java [1]. We describe our Coq implementation of the type soundness proof of FBCF from Section 6 to Section 8. We use version 8.2 of the Coq proof assistant. In order to make this report more readable, we used a pseudo-Coq syntax which slightly differs from the actual Coq syntax from Section 3 to Section 8. The complete Coq files are available on the website http://plrg.kaist.ac.kr/research/software.

2 FBCF Calculus

In this section, we illustrate the syntax and the semantics of FBCF. Section 2.1 presents some assumptions about FBCF. Section 2.2 describes the syntax of FBCF. Section 2.3 and Section 2.4 show the dynamic semantics and the static semantics of FBCF, respectively.

2.1 Assumptions

We make several simplifying assumptions about a FBCF program. These assumptions may be easily checked in a separate phase prior to the type checking phase.

- Each name of a trait or object is unique throughout the program.
- Each name of a method is unique in its defining trait or object.
- Each name of a field is unique in its defining object.
- No trait or object named Object is defined.
- Inheritance relation among traits does not make a cycle.
- Each variable in a type environment Γ is unique.

m			method name
f			field name
x			method parameter name
T			trait name
O			object name
τ	::=	Object	type
	|	T	
	|	O	
e	::=	x	expression
	|	`self`	
	|	$O(\overrightarrow{e})$	
	|	$e\,.\,f$	
	|	$e\,.\,m(\overrightarrow{e})$	
md	::=	$m(\overrightarrow{x:\tau}):\tau = e$	method definition
d	::=	`trait` T `extends` T $\overrightarrow{md}$ `end`	trait or object definition
	|	`object` $O(\overrightarrow{f:\tau})$ `extends` T $\overrightarrow{md}$ `end`	
p	::=	$\overrightarrow{d}\ e$	program

Internal symbols that do not appear in concrete syntax:

C	::=	T	trait or object name
	|	O	
var	::=	x	variable
	|	`self`	
Γ	::=	$\overrightarrow{var:\tau}$	type environment

Figure 1: Syntax of FBCF

2.2 Syntax of FBCF

The syntax of FBCF and the internal symbols that do not appear in the concrete syntax are provided in Figure 1. The syntax of FBCF allows only a small subset of the Fortress language to be formalized. FBCF includes trait and object definitions, field accesses, method invocations, and self expressions. The types of FBCF include traits, objects, and the distinguished trait Object. Note that we syntactically prohibit extending objects. This means that objects should exist as the leaf nodes in the type hierarchy. Among other features of Fortress, FBCF does not include top-level variable and function definitions, overloading, excludes clauses, comprises clauses, where clauses, object expressions, and function expressions.

2.3 Dynamic Semantics of FBCF

The dynamic semantics of FBCF is provided in Figure 2. It presents definitions of a value, evaluation contexts, and redexs as well as evaluation rules. The FBCF dynamic semantics consist of two evaluation rules: one for field accesses and another for method invocations. For simplicity, we use '_' to denote some parts of the syntax that do not have a role in a rule. We assume that does not expand across definition boundaries unless the entire definition is included in it.

2.4 Static Semantics of FBCF

Some auxiliary functions are defined in Figure 3. There are six auxiliary functions: $defined_p$, $inherited_p$, $visible_p$, $mtype_p$, $mbody_p$, and $noOvld$. The $defined_p$ function returns a set of defined method declara-

Value, evaluation contexts and redexes

$$
\begin{array}{lcll}
v & ::= & O(\overrightarrow{v}) & \text{value} \\
E & ::= & \Box & \text{evaluation context} \\
 & | & O(\overrightarrow{e}E\overrightarrow{e}) & \\
 & | & E\,.\,f & \\
 & | & E\,.\,m(\overrightarrow{e}) & \\
 & | & e\,.\,m(\overrightarrow{e}E\overrightarrow{e}) & \\
R & ::= & v\,.\,f & \text{redex} \\
 & | & v\,.\,m(\overrightarrow{v}) &
\end{array}
$$

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

$$
\text{[R-FIELD]} \qquad \frac{\texttt{object}\ O\ \_\ (\overrightarrow{f\colon\ \_})\ \_\ \texttt{end} \in p}{p \vdash E[O(\overrightarrow{v})\,.\,f_i] \longrightarrow E[v_i]}
$$

$$
\text{[R-METHOD]} \qquad \frac{\texttt{object}\ O\ \_\ \texttt{end} \in p \qquad \mathit{mbody}_p(m, O) = \{\overrightarrow{x}\,.\,e\}}{p \vdash E[O(\overrightarrow{v})\,.\,m(\overrightarrow{v'})] \longrightarrow E[[O(\overrightarrow{v})/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x}]e]}
$$

Figure 2: Dynamic Semantics of FBCF

tions in a class, and the $\mathit{inherited}_p$ function returns a set of visible method declarations in an immediate super class of a class that are not overridden. The $\mathit{visible}_p$ function returns every visible method declaration in a class. The mtype_p function returns a set of parameter types and return types of a method. The mbody_p function returns a set of parameter names and method body expressions of a method. The noOvld function guarantees that every method name in a set of method declarations is unique.

The static semantics of FBCF is provided in Figure 4 and Figure 5. The FBCF static semantics is based on the static semantics of Featherweight Java (FJ). The major difference is that FBCF has two kinds of classes, traits and objects. Both trait and object definitions include method definitions but only object definitions include field definitions. However, many of the rules in the FBCF dynamic and static semantics combine the two cases, because of the similarity of traits and objects, Moreover, FBCF allows overriding with the same way that FJ does.

Getting visible methods: $\boxed{\mathit{defined}_p(C)\,/\mathit{inherited}_p(C)\,/\mathit{visible}_p(C) = \{\overrightarrow{(m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}},\ \overrightarrow{x}.\, e)}\}}$

$$\begin{array}{lcl}
\mathit{defined}_p(C) & = & \{(m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}},\ \overrightarrow{x}.\, e) \mid m(\overrightarrow{x\colon \tau_{\mathrm{p}}})\colon \tau_{\mathrm{r}} = e \in \overrightarrow{md}\} \\
 & & \text{where } _\ C\ _\ \overrightarrow{md}\ \texttt{end}\ \in p \\
\mathit{inherited}_p(C) & = & \{(m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}},\ \overrightarrow{x}.\, e) \mid (m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}},\ \overrightarrow{x}.\, e) \in \mathit{visible}_p(T), \\
 & & \qquad \textit{there is no } \tau' \textit{ such that } (m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau',\ _) \in \mathit{defined}_p(C)\} \\
 & & \text{where } _\ C\ \texttt{extends}\ T\ _ \in p \\
\mathit{visible}_p(C) & = & \mathit{defined}_p(C) \cup \mathit{inherited}_p(C)
\end{array}$$

Method type lookup: $\boxed{\mathit{mtype}_p(m, \tau) = \{\overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}}\}}$

$$\begin{array}{lcl}
\mathit{mtype}_p(_, \mathrm{Object}) & = & \emptyset \\
\mathit{mtype}_p(m, C) & = & \{\overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}} \mid (m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}},\ _) \in \mathit{visible}_p(C)\}
\end{array}$$

Method body lookup: $\boxed{\mathit{mbody}_p(m, C) = \{\overrightarrow{x}.\, e\}}$

$$\mathit{mbody}_p(m, C) \quad = \quad \{\overrightarrow{x}.\, e \mid (m,\ \_,\ \overrightarrow{x}.\, e) \in \mathit{visible}_p(C)\}$$

Preventing overloading: $\boxed{p \vdash \mathit{noOvld}(\{\overrightarrow{(m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}}, \overrightarrow{x}.\, e)}\})}$

$$[\textsc{NoOvld}] \quad \frac{\forall (m,\ \_,\ \_) \in S.\ \textit{there is only one element in } S \textit{ which has } m \textit{ as the first component}}{p \vdash \mathit{noOvld}(S)}$$

Figure 3: Auxiliary functions

Program typing: $\boxed{\vdash p : \tau}$

$$\text{[T-PROGRAM]} \quad \frac{p = \overrightarrow{d}\ e \qquad p \vdash \overrightarrow{d}\ \mathsf{ok} \qquad p; \emptyset \vdash e : \tau}{\vdash p : \tau}$$

Trait or object definition typing: $\boxed{p \vdash d\ \mathsf{ok}}$

$$\text{[T-TRAITDEF]} \quad \frac{p \vdash T'\ \mathsf{ok} \qquad p; \texttt{self} : T; T \vdash \overrightarrow{md}\ \mathsf{ok} \qquad p \vdash \mathit{noOvld}(\mathit{visible}_p(T))}{p \vdash \texttt{trait}\ T\ \texttt{extends}\ T'\ \overrightarrow{md}\ \texttt{end}\ \mathsf{ok}}$$

$$\text{[T-OBJECTDEF]} \quad \frac{p \vdash T\ \mathsf{ok} \qquad p \vdash \overrightarrow{\tau_{\mathrm{f}}}\ \mathsf{ok} \qquad p; \texttt{self} : O\ \overrightarrow{f : \tau_{\mathrm{f}}}; O \vdash \overrightarrow{md}\ \mathsf{ok} \qquad p \vdash \mathit{noOvld}(\mathit{visible}_p(O))}{p \vdash \texttt{object}\ O(\overrightarrow{f : \tau_{\mathrm{f}}})\ \texttt{extends}\ T\ \overrightarrow{md}\ \texttt{end}\ \mathsf{ok}}$$

Method definition typing: $\boxed{p; \Gamma; C \vdash md\ \mathsf{ok}}$

$$\text{[T-METHODDEF]} \quad \frac{\_\, C\, \_\ \texttt{extends}\ T\ \_ \in p \qquad p \vdash \mathit{override}(m, T, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}}) \qquad p \vdash \overrightarrow{\tau_{\mathrm{p}}}\ \mathsf{ok} \qquad p \vdash \tau_{\mathrm{r}}\ \mathsf{ok} \qquad p; \Gamma \overrightarrow{x : \tau_{\mathrm{p}}} \vdash e : \tau \qquad p \vdash \tau\ <:\ \tau_{\mathrm{r}}}{p; \Gamma; C \vdash m(\overrightarrow{x : \tau_{\mathrm{p}}}) : \tau_{\mathrm{r}} = e\ \mathsf{ok}}$$

Method overriding: $\boxed{p \vdash \mathit{override}(m, T, \overrightarrow{\tau} \to \tau)}$

$$\text{[OVERRIDING]} \quad \frac{\mathit{mtype}_p(m, T) = \{\overrightarrow{\tau'_{\mathrm{p}}} \to \tau'_{\mathrm{r}}\} \qquad \overrightarrow{\tau'_{\mathrm{p}}} = \overrightarrow{\tau_{\mathrm{p}}} \qquad p \vdash \tau_{\mathrm{r}}\ <:\ \tau'_{\mathrm{r}}}{p \vdash \mathit{override}(m, T, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}})}$$

Figure 4: Static Semantics of FBCF (I)

Expression typing: $\boxed{p; \Gamma \vdash e : \tau}$

[T-VAR] $$\frac{var \in dom(\Gamma)}{p; \Gamma \vdash var : \Gamma(var)}$$

[T-OBJECT] $$\frac{\texttt{object}\ O(\overrightarrow{\_ : \tau_{\mathrm{f}}})\ \_\ \texttt{end} \in p \qquad p; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau} \qquad p \vdash \overrightarrow{\tau}\ <:\ \overrightarrow{\tau_{\mathrm{f}}}}{p; \Gamma \vdash O(\overrightarrow{e}) : O}$$

[T-FIELD] $$\frac{p; \Gamma \vdash e_{\mathrm{o}} : O \qquad \texttt{object}\ O(\overrightarrow{f : \tau})\ \_\ \texttt{end} \in p}{p; \Gamma \vdash e_{\mathrm{o}}\,.\,f_i : \tau_i}$$

[T-METHOD] $$\frac{p; \Gamma \vdash e_{\mathrm{o}} : \tau_{\mathrm{o}} \qquad \mathit{mtype}_p(m, \tau_{\mathrm{o}}) = \{\overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}}\} \qquad p; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau} \qquad p \vdash \overrightarrow{\tau}\ <:\ \overrightarrow{\tau_{\mathrm{p}}}}{p; \Gamma \vdash e_{\mathrm{o}}\,.\,m(\overrightarrow{e}) : \tau_{\mathrm{r}}}$$

Subtyping: $\boxed{p \vdash \tau\ <:\ \tau}$

[S-OBJ] $$p \vdash \tau\ <:\ \mathrm{Object}$$

[S-REFL] $$p \vdash \tau\ <:\ \tau$$

[S-TRANS] $$\frac{p \vdash \tau_1\ <:\ \tau_2 \qquad p \vdash \tau_2\ <:\ \tau_3}{p \vdash \tau_1\ <:\ \tau_3}$$

[S-TAPP] $$\frac{\_\ C\ \texttt{extends}\ T\ \_ \in p}{p \vdash C\ <:\ T}$$

Well formed types: $\boxed{p \vdash \tau\ \mathsf{ok}}$

[W-OBJ] $p \vdash \mathrm{Object}\ \mathsf{ok}$

[W-TAPP] $$\frac{\_\ C\ \_ \in p}{p \vdash C\ \mathsf{ok}}$$

Figure 5: Static Semantics of FBCF (II)

3 Atom

Support for atoms, i.e., objects with decidable equality. We provide here the ability to generate an atom fresh for any finite collection, i.e., the lemma *atom_fresh_for_set*.

Original authors: Arthur Chargueraud and Brian Aydemir.

This section was taken from Cast-Free Featherweight Java [1].

```
Require Import List.
Require Import Max.
Require Import Le.
Require Peano_dec.
```

3.1 Definition

Atoms are structureless objects such that we can always generate one fresh from a finite collection. Equality on atoms is *eq* and decidable. We use Coq's module system to make abstract the implementation of atoms. The `Export` *AtomImpl* line below allows us to refer to the type *atom* and its properties without having to qualify everything with "*AtomImpl*.".

```
Module Type ATOM.

  Parameter atom : Set.

  Parameter atom_fresh_for_list :
    ∀ (xs : list atom), {x : atom | ¬ List.In x xs}.

  Parameter eq_atom_dec : ∀ x y : atom, {x = y} + {x ≠ y}.

End ATOM.
```

The implementation of the above interface is hidden for documentation purposes.

```
Module AtomImpl : ATOM.

End AtomImpl.

Export AtomImpl.
```

4 AdditionalTactics

```
Require Export String.
Open Scope string_scope.
```

4.1 Delineating Cases in Proofs

This section was taken from the POPLmark Wiki [4] and Cast-Free Featherweight Java [1].

4.1.1 Tactic Definitions

```
Ltac move_to_top x :=
  match reverse goal with
  | H : _ ⊢ _ ⇒ try move x after H
  end.

Tactic Notation "assert_eq" ident(x) constr(v) :=
  let H := fresh in
  assert (x = v) as H by reflexivity;
  clear H.
```

```
Tactic Notation "Case_aux" ident(x) constr(name) :=
  first [
    set (x := name); move_to_top x
  | assert_eq x name
  | fail 1 "because we are working on a different case." ].

Ltac Case name := Case_aux case name.
Ltac SCase name := Case_aux subcase name.
Ltac SSCase name := Case_aux subsubcase name.
```

4.2 Do Some Automatic Destructions

```
Tactic Notation "decompose" "records" :=
repeat (match goal with
| H: _ ⊢ _ ⇒ progress (decompose record H); clear H
end).

Tactic Notation "subst" "injections" :=
repeat (progress (match goal with
| H: _ = _ ⊢ _ ⇒ injection H; clear H; intros; subst
end)).

Tactic Notation "destructs" hyp(H) := try red in H; decompose record H.

Tactic Notation "forward" hyp(H) "by" tactic(t) :=
    lapply H; [ clear H; intro H | t ].

Tactic Notation "forward" hyp(H) := forward H by idtac.

Tactic Notation "contradiction" "by" tactic(t) :=
  assert False; [ t | contradiction ].

Tactic Notation "auto" "via" constr(e) :=
    let H := fresh in (assert (H: e) by auto; try red in H; try decompose record H;
auto).

Tactic Notation "eauto" "via" constr(e) :=
    let H := fresh in (assert (H: e) by eauto; try red in H; try decompose record H;
eauto).

Tactic Notation "specialize" "trivial" :=
repeat match goal with
| H: ?H0 → _ ⊢ _ ⇒
  let HT := fresh in
  (assert (HT: H0) by trivial;
  specialize (H HT);
  clear HT)
end.

Tactic Notation "fold_simpl" constr(t) :=
    let et := (eval simpl in t) in change et with t.
```

5 Metatheory

This library provides a number of auxiliary constructs (and their properties) for the study of programming languages in Coq. The definition of these constructs is mostly straightforward.

This section was taken from Cast-Free Featherweight Java [1].

```
Require Import AdditionalTactics.
```

```
Require Export Atom.
Require Export List.
Set Implicit Arguments.
```

5.1 Atoms

```
Notation "x ≐ y" := (eq_atom_dec x y) (at level 67).
Fact eq_atom_true: ∀ (A : Type) a (c d : A),
    (if a ≐ a then c else d) = c.
Proof.
  intros.
  destruct (a ≐ a); [ | destruct n]; reflexivity.
Qed.
Fact eq_atom_false: ∀ (A : Type) a b (c d : A),
    a ≠ b → (if a ≐ b then c else d) = d.
Proof.
  intros.
  destruct (a ≐ b); [ subst; destruct H | ]; reflexivity.
Qed.
Definition In_atom_list_dec := In_dec eq_atom_dec.
```

5.2 Environments

An environment maps atoms to some value of an variable type A. We model an environment as a list of pairs: *list* (*atom* × A).

```
Notation "x ∈ E" := (In x E) (at level 69).
Notation "x ∉ E" := (¬ In x E) (at level 69).
Section Environment.
    Variable A: Type.
```

The function *get x E* retrieves the first binding of x in environment E.

```
    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.
```

binds x v E holds when x is bound to v in E. *no_binds x E* holds when there is no binding for x in E.

```
    Definition binds x v E : Prop :=
        get x E = Some v.
    Definition no_binds x E : Prop :=
        get x E = None.
    Fact binds_first: ∀ x (a : A) E, binds x a ((x,a)::E).
    Proof.
      intros.
      unfold binds. simpl.
      apply eq_atom_true.
    Qed.
    Fact binds_other: ∀ x y (a b : A) E,
        binds y b E → x ≠ y → binds y b ((x,a)::E).
```

```
Proof.
  intros.
  unfold binds in ×. simpl.
  rewrite eq_atom_false with (1:=sym_not_eq H0).
  assumption.
Qed.

Fact binds_nil: ∀ x (a : A), ~binds x a nil.
Proof.
  unfold binds. simpl. intros. discriminate.
Qed.

Fact binds_fun: ∀ E x (a b : A),
    binds x a E → binds x b E → a = b.
Proof.
  unfold binds. intros E x a b H H'.
  rewrite H' in H. injection H.
  auto.
Qed.

Fact binds_elim_eq: ∀ x (a b : A) E,
    binds x a ((x,b)::E) → a = b.
Proof.
  intros x a b E.
  unfold binds. simpl.
  rewrite eq_atom_true.
  injection 1. auto.
Qed.

Fact binds_elim_neq: ∀ x y (a b : A) E,
    x ≠ y → binds x a ((y,b)::E) → binds x a E.
Proof.
  intros x y a b E H.
  unfold binds. simpl.
  rewrite eq_atom_false; trivial.
Qed.

Fact binds_head : ∀ x a E F,
    binds x a F → binds x a (F ++ E).
Proof.
  unfold binds.
  induction F as [|(y,b)]; simpl; intros H.
  Case "nil". discriminate.
  Case "cons". destruct (x ≐ y); auto.
Qed.

Fact binds_nobinds : ∀ x a E,
    binds x a E → no_binds x E → False.
Proof.
  unfold binds. unfold no_binds. intros.
  rewrite H in H0. discriminate.
Qed.

Fact nobinds_nil: ∀ x, no_binds x nil.
Proof.
  intros.
  unfold no_binds. reflexivity.
```

```
Qed.

Fact nobinds_cons: ∀ x y b E,
     no_binds x E →
     x ≠ y →
     no_binds x ((y,b)::E).
Proof.
  unfold no_binds. intros.
  simpl.
  rewrite eq_atom_false with (1:=H0).
  assumption.
Qed.

Fact nobinds_app: ∀ x E1 E2,
    no_binds x E1 →
    no_binds x E2 →
    no_binds x (E1++E2).
Proof.
  unfold no_binds. intros.
  induction E1 as [|(y,v)]; simpl.
  Case "nil". assumption.
  Case "cons". destruct (x==y).
    SCase "x=y".
      subst. rewrite binds_first in H. discriminate H.
    SCase "x<>y".
      simpl in H. rewrite eq_atom_false with (1:=n) in H.
      auto.
Qed.
```

The functions *keys E* and *dom E* retrieve the atoms that are bound in the environment *E*.

```
Definition keys (E: list (atom × A)) : list atom :=
    List.map (fun p ⇒ match p with (x,_) ⇒ x end) E.

Definition dom := keys.

Fact dom_binds: ∀ (E : list (atom × A)) (x : atom),
    x ∈ dom E → ∃ v:A, binds x v E.
Proof.
  intros.
  induction E as [| (u, v) E]; simpl in H.
  Case "nil". contradiction.
  Case "cons".
    unfold binds in ×. simpl in ×.
    destruct (x ≟ u); [ eauto | apply IHE ].
    destruct H; [ symmetry in H; contradiction | apply H ].
Qed.

Fact binds_In : ∀ a x E,
    binds x a E → x ∈ dom E.
Proof.
  intros.
  induction E as [| (u, v) E].
  Case "nil". contradiction (binds_nil H).
  Case "cons".
    unfold binds in ×. simpl in ×.
    destruct (x ≟ u); auto.
```

```
Qed.
Fact dom_binds_neg: ∀ (E : list (atom × A)) (x : atom),
    x ∉ dom E → get x E = None.
Proof.
  intros.
  induction E as [| (u, v) E].
  Case "nil". reflexivity.
  Case "cons".
    simpl in ×.
    destruct (x ≐ u); [ contradiction H | apply IHE ]; auto.
Qed.
```

The function *imgs E* retrieves the values in *E*. *map f E* applies *f* to the values in *E*.

```
Definition imgs (E: list (atom × A)) : list A :=
    List.map (fun p ⇒ match p with (_,v) ⇒ v end) E.
Definition map (B: Type) (f: A → B) (E: list (atom × A)) : list (atom × B) :=
    List.map (fun p ⇒ match p with (x,v) ⇒ (x, f v) end) E.
```

ok E holds when the environment *E* contains no duplicate bindings.

```
Inductive ok: list (atom × A) → Prop :=
| ok_nil: ok nil
| ok_cons: ∀ E x v, ok E → no_binds x E → ok ((x,v)::E).
Fact binds_concat_ok : ∀ x a E F,
  binds x a E → ok (F ++ E) → binds x a (F ++ E).
Proof.
  induction F as [|(y,b)]; simpl; intros H Ok.
  Case "nil". assumption.
  Case "cons".
    inversion Ok. subst.
    destruct (y ≐ x).
    SCase "x=y".
      subst.
      contradiction binds_nobinds with (a:=a) (2:=H4).
      eauto.
    SCase "x<>y".
      apply binds_other; [ auto | assumption ].
Qed.
Fact binds_weaken : ∀ x a E F G,
  binds x a (G ++ E) →
  ok (G ++ F ++ E) →
  binds x a (G ++ F ++ E).
Proof.
  induction G as [|(y,b)]; simpl; intros H Ok.
  Case "nil". apply binds_concat_ok; assumption.
  Case "cons".
    inversion Ok. subst.
    destruct (y ≐ x).
    SCase "x=y".
      subst.
      rewrite binds_elim_eq with (1:=H).
      apply binds_first.
    SCase "x<>y".
```

```
    apply binds_other with (2:=n).
    apply IHG; [ eauto using binds_elim_neq | assumption ].
Qed.
```

forall_env P E holds when proposition *P x v* holds for all bindings (*x*, *v*) in environment *E*.

```
Variable P: atom → A → Prop.

Inductive forall_env : list (atom × A) → Prop :=
| fa_nil: forall_env nil
| fa_cons: ∀ E x v, forall_env E → P x v → forall_env ((x,v)::E).

Hint Constructors forall_env.

Fact fa_single: ∀ x a, P x a → forall_env ((x,a)::nil).
Proof.
  auto.
Qed.

Fact fa_app: ∀ E F,
    forall_env E → forall_env F → forall_env (E++F).
Proof.
  intros; induction H; simpl; auto.
Qed.

Fact fa_weaken: ∀ E F G,
    forall_env (E++F++G) → forall_env (E++G).
Proof.
  intros; induction E as [| (y,a)]; simpl.
  Case "nil".
    induction F as [| (y,a)].
    SCase "nil". trivial.
    SCase "cons". apply IHF. inversion H. subst. assumption.
  Case "cons". inversion H. subst. auto.
Qed.

Fact fa_binds_elim: ∀ x a E,
    binds x a E → forall_env E → P x a.
Proof.
  intros; induction H0.
  Case "fa_nil". contradiction (binds_nil H).
  Case "fa_cons". destruct (x ≐ x0).
    SCase "x = x0". subst. rewrite (binds_elim_eq H). assumption.
    SCase "x ≠ x0". eauto using binds_elim_neq.
Qed.

End Environment.

Unset Implicit Arguments.
Hint Constructors ok.
Hint Constructors forall_env.
Implicit Arguments fa_nil [A].

Set Implicit Arguments.
```

env_zip aenv bs benv will match up environment *aenv* with the value list *bs* to produce the environment *benv*.

```
Inductive env_zip (A: Type) (B: Type) : list (atom × A) → list B → list (atom × B) → Prop
:=
| z_nil : env_zip nil nil nil
```

| z_cons : ∀ *x a b aenv bs benv*,
env_zip *aenv bs benv* →
env_zip ((*x*,a)::aenv) (*b*::bs) ((*x*,b)::benv).

`Hint` *Constructors env_zip*.

`Unset Implicit` *Arguments*.

6 FBCF Definitions

This library contains the actual definitions of the FBCF syntax, auxiliary functions, dynamic semantics, and static semantics.

`Require Import` Metatheory.

6.1 Syntax

6.1.1 Lexical Categories

Names of variables, fields, methods, objects, and traits are atoms; their equality is decidable.

`Definition` var := *atom*.
`Definition` fname := *atom*.
`Definition` mname := *atom*.
`Definition` oname := *atom*.
`Definition` tname := *atom*.

The names *self* and *Object* are predefined. We simply assume that these names exist.

`Parameter` *self* : var.
`Parameter` *Object* : tname.

6.1.2 Types and Expressions

A type is either a trait or an object.

`Definition` ttyp := tname.
`Definition` otyp := oname.

`Inductive` **typ** : `Set` :=
| ttyp2typ : ttyp → **typ**
| otyp2typ : otyp → **typ**.

Expression forms are variable references (*self* and variable names), object constructions, field accesses, and method invocations.

`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**.

6.1.3 Environments and Class Tables

env declares a mapping between variables and their types. *benv* binds variables to expressions.

`Notation` env := (**list** (var × **typ**)).
`Notation` benv := (**list** (var × **exp**)).

flds and *mths* map the names of fields and methods to their definitions.

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

otable maps the names of objects to their definitions. An object definition consists of its name, supertrait's name, and the fields and methods defined in it.

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

ttable maps the names of traits to their definitions. A trait definition consists of its name, supertrait's name, and the methods defined in it. A trait definition does not have any field declarations.

```
Notation ttable := (list (tname × (tname × mths))).
```

We assume a fixed object table *OT*, a fixed trait table *TT*, and a fixed expression *Ex* for a program.

```
Parameter OT : otable.
Parameter TT : ttable.
Parameter Ex : exp.
Notation Program := (OT × TT × Ex).
```

6.2 Auxiliary Functions

6.2.1 Field and Method Lookup

tr_fields T flds holds if a set of field declarations *flds* is defined in a trait *T* in the class hierarchy; *flds* should be *nil* because traits should not have any fields. *ob_fields O flds* holds if a set of field declarations *flds* is defined in an object *O* in the class hierarchy.

```
Inductive tr_fields : tname → flds → Prop :=
  | fields_Object : tr_fields Object nil
  | fields_trt : ∀ T T' ms,
      binds T (T', ms) TT →
      tr_fields T nil.

Inductive ob_fields : oname → flds → Prop :=
  | fields_obj : ∀ O T fs ms,
      binds O (T, fs, ms) OT →
      ob_fields O fs.

Hint Constructors tr_fields ob_fields.
```

field O f t holds if a field named *f* with a type *t* is defined in an object *O*.

```
Definition field (O : oname) (f : fname) (t : typ) : Prop :=
  exists2 fs, ob_fields O fs & binds f t fs.

Hint Unfold field.
```

tr_method T m mdecl holds if a method named *m* with a declaration *mdecl* is defined in a trait *T* in the class hierarchy. *ob_method O m mdecl* holds if a method named *m* with a declaration *mdecl* is defined in an object *O* in the class hierarchy.

```
Inductive tr_method : tname → mname → env*typ*exp → Prop :=
  | method_self_trt : ∀ T T' ms m mdecl,
      binds T (T', ms) TT →
      binds m mdecl ms →
      tr_method T m mdecl
  | method_super_trt : ∀ T T' ms m mdecl,
      binds T (T', ms) TT →
      no_binds m ms →
      tr_method T' m mdecl →
      tr_method T m mdecl.
```

```
Inductive ob_method : oname → mname → env*typ*exp → Prop :=
    | method_self_obj : ∀ O T fs ms m mdecl,
        binds O (T, fs, ms) OT →
        binds m mdecl ms →
        ob_method O m mdecl
    | method_super_obj : ∀ O T fs ms m mdecl,
        binds O (T, fs, ms) OT →
        no_binds m ms →
        tr_method T m mdecl →
        ob_method O m mdecl.

Hint Constructors tr_method ob_method.
```

6.2.2 Expression Substitution

subst_exp E e returns an expression *e* where any occurrence of bound variables has been replaced by its binding in an environment *E*.

```
Fixpoint subst_exp (E : benv) (e : exp) {struct e} : exp :=
    match e with
    | e_var v ⇒
        match get v E with
        | Some e' ⇒ e'
        | None ⇒ e_var v
        end
    | e_self self ⇒
        match get self E with
        | Some e' ⇒ e'
        | None ⇒ e_self self
        end
    | e_new C es ⇒ e_new C (List.map (subst_exp E) es)
    | e_field e' f' ⇒ e_field (subst_exp E e') f'
    | e_method e' m' es'
        ⇒ e_method (subst_exp E e') m' (List.map (subst_exp E) es')
    end.
```

6.3 Dynamic Semantics

6.3.1 Values

```
Inductive val : exp → Prop :=
    | val_new : ∀ on es,
        (∀ e, In e es → val e) → val (e_new on es).

Hint Constructors val.
```

6.3.2 Evaluation Contexts

We model evaluation contexts as a function of type *exp* → *exp*. *exp_context EE* holds if *EE* is an evaluation context. Basically, any subexpression of an expression is an evaluation context. This means that the evaluation semantics of FBCF is nondeterministic.

```
Inductive exps_context : (exp → list exp) → Prop :=
    | esc_head : ∀ es,
        exps_context (fun e ⇒ e::es)
    | esc_tail : ∀ e EE,
```

exps_context *EE* →
exps_context (fun *e0* ⇒ *e*::(EE *e0*)).

Inductive **exp_context** : (**exp** → **exp**) → Prop :=
| ec_null :
exp_context (fun *e* ⇒ *e*)
| ec_new_args : ∀ *C EE* ,
exps_context *EE* →
exp_context (fun *e* ⇒ e_new *C* (*EE e*))
| ec_field : ∀ *f*,
exp_context (fun *e0* ⇒ e_field *e0 f*)
| ec_meth : ∀ *m es*,
exp_context (fun *e0* ⇒ e_method *e0 m es*)
| ec_meth' : ∀ *m e0 EE*,
exps_context *EE* →
exp_context (fun *e* ⇒ e_method *e0 m* (*EE e*)).

Hint *Constructors exp_context exps_context*.

6.3.3 Redexes

We consider redexes in the evaluation rules.

6.3.4 Evaluation Rules

eval_rule eval_context redex eval_context exp holds when an expression *e* reduces to another expression *e'* in one step.

Inductive **eval_rule** : **exp** → **exp** → Prop :=
| r_field : ∀ *O fs es f e fes*,
ob_fields *O fs* →
env_zip *fs es fes* →
binds *f e fes* →
eval_rule (e_field (e_new *O es*) *f*) *e*
| r_method : ∀ *O m E t e es ves es0*,
ob_method *O m* (*E*, *t*, *e*) →
env_zip *E es ves* →
eval_rule (e_method (e_new *O es0*) *m es*) (subst_exp ((*self*, e_new *O es0*)::ves) *e*)
| r_context : ∀ *EE e e'*,
eval_rule *e e'* →
exp_context *EE* →
eval_rule (*EE e*) (*EE e'*).

Hint *Constructors eval_rule*.

Hint Extern 2 (**eval_rule** (e_field _ ?f) _) ⇒ eapply (r_context (fun *e0* ⇒ e_field *e0 f*)).
Hint Extern 2 (**eval_rule** (e_method _ ?m ?es) _) ⇒ eapply (r_context (fun *e0* ⇒ e_method *e0 m es*)).
Hint Extern 2 (**eval_rule** (e_method ?e0 ?m (?EE _)) _) ⇒ eapply (r_context (fun *e* ⇒ e_method *e0 m* (*EE e*))).
Hint Extern 2 (**eval_rule** (e_new ?C (?EE _)) _) ⇒ eapply (r_context (fun *e* ⇒ e_new *C* (*EE e*))).

6.4 Static Semantics

6.4.1 Well-formed Types

wf_type t holds when a type *t* is well formed.

```
Inductive wf_type : typ → Prop :=
    | wf_obj : wf_type (ttyp2typ Object)
    | wf_tapp_obj : ∀ O, (O ∈ dom OT) → wf_type (otyp2typ O)
    | wf_tapp_trt : ∀ T, (T ∈ dom TT) → wf_type (ttyp2typ T).

Hint Constructors wf_type.
```

6.4.2 Subtyping

ob_extends O T holds if an object *O* is an immediate subtype of a trait *T*. Note that objects cannot be a supertype of any type. *tr_extends T T'* holds if a trait *T* is an immediate subtype of a trait *T'*.

```
Definition ob_extends (O : oname) (T : tname) : Prop :=
    ∃ fs, ∃ ms, (binds O (T, fs, ms) OT).

Definition tr_extends (T : tname) (T' : tname) : Prop :=
    ∃ ms, (binds T (T', ms) TT).

Hint Unfold ob_extends tr_extends.
```

sub_ty t1 t2 holds if *t1* is a subtype of *t2*. The subtype relation is a reflexive and transitive closure of the extends relation.

```
Inductive sub_ty : typ → typ → Prop :=
    | s_obj : ∀ t, sub_ty t (ttyp2typ Object)
    | s_refl : ∀ t, sub_ty t t
    | s_tran : ∀ t1 t2 t3, sub_ty t1 t2 → sub_ty t2 t3 → sub_ty t1 t3
    | s_ob_tapp : ∀ O T, ob_extends O T → sub_ty (otyp2typ O) (ttyp2typ T)
    | s_tr_tapp : ∀ T T', tr_extends T T' → sub_ty (ttyp2typ T) (ttyp2typ T').

Hint Constructors sub_ty.
```

6.4.3 Expression Typing

exp_ty E e t holds when an expression *e* has a type *t* in an environment *E*. *wide_typing E e t* holds when an expression *e* has a subtype of a type *t*. *wide_typings E es ts* holds when each expression *e* in *es* has a subtype of corresponding *t* in *ts*.

```
Inductive exp_ty : env → exp → typ → Prop :=
    | t_var : ∀ x t E,
        ok E →
        binds x t E →
        exp_ty E (e_var x) t
    | t_self : ∀ t E,
        ok E →
        binds self t E →
        exp_ty E (e_self self) t
    | t_object : ∀ E O fs es,
        ob_fields O fs →
        wf_type (otyp2typ O) →
        wide_typings E es (imgs fs) →
        exp_ty E (e_new O es) (otyp2typ O)
    | t_field : ∀ E e0 O t f,
        exp_ty E e0 (otyp2typ O) →
```

```
        field O f t →
        exp_ty E (e_field e0 f) t
    | t_tr_method : ∀ E e0 O m ME t0' body es,
        exp_ty E e0 (ttyp2typ O) →
        tr_method O m (ME, t0', body) →
        wide_typings E es (imgs ME) →
        exp_ty E (e_method e0 m es) t0'
    | t_ob_method : ∀ E e0 O m ME t0' body es,
        exp_ty E e0 (otyp2typ O) →
        ob_method O m (ME, t0', body) →
        wide_typings E es (imgs ME) →
        exp_ty E (e_method e0 m es) t0'
with wide_typing : env → exp → typ → Prop :=
    | wt_sub : ∀ E e t t',
        exp_ty E e t →
        sub_ty t t' →
        wide_typing E e t'
with wide_typings : env → list exp → list typ → Prop :=
    | wts_nil : ∀ E, ok E → wide_typings E nil nil
    | wts_cons : ∀ E E0 es e t,
        wide_typings E es E0 →
        wide_typing E e t →
        wide_typings E (e::es) (t::E0).

Hint Constructors exp_ty wide_typing wide_typings.
```

6.4.4 Declaration Typing

meth_ob_ty O T m E t e holds when (*E*, *t*, *e*) is a valid method declaration for a method *m* in an object *O* with a supertrait *T*. *meth_tr_ty T T' m E t e* holds when (*E*, *t*, *e*) is a valid method declaration for a method *m* in a trait *T* with a supertrait *T'*. FBCF supports method overriding when an overriding method in a subtype has the same number and the same types of the parameters with the overridden method in a supertrait, and the return type of the overriding method is a subtype of the return type of the overridden method.

```
Definition meth_overriding (m : mname) (T : tname) (E : env) (t : typ): Prop :=
    ∀ E' t' e,
    tr_method T m (E', t', e) →
    sub_ty t t' ∧ imgs E = imgs E'.

Hint Unfold meth_overriding.

Definition meth_ob_ty (O : oname) (T : tname) (m : mname) (E : env) (t0 : typ) (e : exp) :
Prop :=
    meth_overriding m T E t0 ∧ wide_typing ((self, (otyp2typ O))::E) e t0.

Definition meth_tr_ty (T T' : tname) (m : mname) (E : env) (t0 : typ) (e : exp) : Prop :=
    meth_overriding m T' E t0 ∧ wide_typing ((self, (ttyp2typ T))::E) e t0.

Hint Unfold meth_ob_ty meth_tr_ty.

Definition meth_ob_ty' (O : oname) (T : tname) (m : mname) (sig : env*typ*exp) : Prop :=
    match sig with (E, t, e) ⇒ meth_ob_ty O T m E t e end.

Definition meth_tr_ty' (T T' : tname) (m : mname) (sig : env*typ*exp) : Prop :=
    match sig with (E, t, e) ⇒ meth_tr_ty T T' m E t e end.

Hint Unfold meth_ob_ty' meth_tr_ty'.
```

def_ob_ty O T fs ms holds when an object *O* has a supertrait *T*, fields *fs*, and methods *ms*. *def_tr_ty T T' ms* holds when a trait *T* has a supertrait *T'* and methods *ms*.

```
Definition def_ob_ty (O : oname) (T : tname) (fds : flds) (ms : mths) : Prop :=
  ok fds ∧ ok ms ∧ forall_env (meth_ob_ty' O T) ms.
Definition def_tr_ty (T T' : tname) (ms : mths) : Prop :=
  ok ms ∧ forall_env (meth_tr_ty' T T') ms.
Hint Unfold def_ob_ty def_tr_ty.
Definition def_ob_ty' (O : oname) (sig : tname × flds × mths) : Prop :=
  match sig with (T, flds, mths) ⇒ def_ob_ty O T flds mths end.
Definition def_tr_ty' (T : tname) (sig : tname × mths) : Prop :=
  match sig with (T', mths) ⇒ def_tr_ty T T' mths end.
Hint Unfold def_ob_ty' def_tr_ty'.
```

The *prog_ty* holds when whole of the program is valid.

```
Definition ok_otable ot := ok ot ∧ forall_env def_ob_ty' ot.
Definition ok_ttable tt := ok tt ∧ forall_env def_tr_ty' tt.
Hint Unfold ok_otable ok_ttable.
Definition Prog_ty : Prop :=
  ok_otable OT ∧ ok_ttable TT ∧ ∃ t, exp_ty nil Ex t.
Hint Unfold Prog_ty.
```

The following module defines the hypotheses about the safety argument. We assume that *Object* is not defined in the object table *OT* and the trait table *TT* if the program is well formed.

```
Module Type HYPS.
  Parameter ot_noobj : Object ∉ dom OT.
  Parameter tt_noobj : Object ∉ dom TT.
  Parameter ok_prog : Prog_ty.
  Parameter ok_ot: ok_otable OT.
  Parameter ok_tt: ok_ttable TT.
End HYPS.
```

We prove the type soundness of FBCF by implementing the following module type: given the above hypotheses, we prove the preservation and progress theorems.

```
Module Type SAFETY (H: HYPS).
  Parameter preservation : ∀ E e e' t,
    exp_ty E e t →
    eval_rule e e' →
    wide_typing E e' t.
  Parameter progress : ∀ e t,
    exp_ty nil e t →
    val e ∨ (∃ e', eval_rule e e').
End SAFETY.
```

7 FBCF Facts

In this library, we prove several properties that are helpful to prove the type soundness of FBCF. Some properties may not be needed for a hand proof.

```
Require Import AdditionalTactics.
Require Import Metatheory.
Require Import FBCF_Definition.
```

7.1 Auxiliary Facts

7.1.1 Mutual Induction Schemes for Expression Typing

```
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.*

7.1.2 Inversions to Retrieve Information from Typing

```
Fact typings_implies_ok_env:
    (∀ E e t, exp_ty E e t → ok E) ∧
    (∀ E e t, wide_typing E e t → ok E) ∧
    (∀ E es env, wide_typings E es env → ok E).
Proof.
  apply typings_mutind; intros; trivial.
Qed.

Definition typing_implies_ok_env := proj1 typings_implies_ok_env.
Definition wide_typing_implies_ok_env := proj1 (proj2 typings_implies_ok_env).
Definition wide_typings_implies_ok_env := proj2 (proj2 typings_implies_ok_env).

Hint Immediate typing_implies_ok_env wide_typing_implies_ok_env wide_typings_implies_ok_env.

Fact ok_ob_class_meth: ∀ O T fs ms m t E e,
    def_ob_ty O T fs ms →
    binds m (E, t, e) ms →
    meth_ob_ty O T m E t e.
Proof.
  intros.
  destruct H.
  fold (meth_ob_ty' O T m (E, t, e)).
  destruct H1.
  eapply fa_binds_elim; eassumption.
Qed.

Fact ok_tr_class_meth: ∀ T T' ms m t E e,
    def_tr_ty T T' ms →
    binds m (E, t, e) ms →
    meth_tr_ty T T' m E t e.
Proof.
    intros.
    destruct H.
    fold (meth_tr_ty' T T' m (E, t, e)).
    eapply fa_binds_elim; eassumption.
Qed.

Fact ok_otable_class: ∀ ot O T fs ms,
    ok_otable ot →
    binds O (T, fs, ms) ot →
    def_ob_ty O T fs ms.
Proof.
  intros.
  destruct H as (_, H).
  fold (def_ob_ty' O (T, fs, ms)).
```

```
  eapply fa_binds_elim; eassumption.
Qed.

Fact ok_ttable_class: ∀ tt T T' ms,
    ok_ttable tt →
    binds T (T', ms) tt →
    def_tr_ty T T' ms.
Proof.
  intros.
  destruct H as (_ , H).
  fold (def_tr_ty' T (T', ms)).
  eapply fa_binds_elim; eassumption.
Qed.
```

7.1.3 Correspondence between Keys and Images of the Same Environment

```
Fact wide_typings_implies_zip: ∀ E0 E ds,
    wide_typings E0 ds (imgs E) →
    ∃ Eds, env_zip E ds Eds.
Proof.
  intros. generalize dependent ds.
  induction E; intros; inversion H; subst.
  Case "el_nil". eauto.
  Case "el_cons".
    destruct IHE with (1 := H4).
    destruct a. eauto.
Qed.

Fact binds_zip: ∀ E0 E ds Eds v t,
    wide_typings E0 ds (imgs E) →
    env_zip E ds Eds →
    binds v t E →
    (exists2 e, binds v e Eds & wide_typing E0 e t).
Proof.
  intros E0 E ds Eds v t. revert ds Eds.
  induction E; intros; [ | destruct a as (a0,t0)]; inversion H1.
  Case "cons".
    inversion H0. subst. inversion H. subst.
    destruct (v == a0).
    SCase "v = a0". subst injections. eauto using binds_first.
    SCase "v ≠ a0".
      destruct IHE with (1:=H7) (2:=H8) (3:=H3).
      eauto using binds_other.
Qed.
```

7.2 Properties and Generalized Typing Rules

The generalized typing rules are similar to the ordinary expression typing rules but they require *wide_typing* instead of *typing* in their conditions. They usually follow from such properties as subtyping preserves methods and fields.

```
Fact tr_method_fun: ∀ T m mdecl mdecl',
    tr_method T m mdecl → tr_method T m mdecl' → mdecl = mdecl'.
Proof.
```

```
  intros T m mdecl mdecl' H H'.
  induction H; destruct H'; assert ((T', ms) = (T'0, ms0))
    by (eapply binds_fun; eassumption); subst injections; try eauto.
  Case "method_self-method_self".
    eapply binds_fun; eassumption.
  Case "method_self-method_other".
    contradiction by (eauto using binds_nobinds).
  Case "method_other-method_self".
    contradiction by (eauto using binds_nobinds).
Qed.

Fact ob_method_fun: ∀ O m mdecl mdecl',
    ob_method O m mdecl → ob_method O m mdecl' → mdecl = mdecl'.
Proof.
  intros T m mdecl mdecl' H H'.
  induction H; destruct H'; assert ((T, fs, ms) = (T0, fs0, ms0))
    by (eapply binds_fun; eassumption); subst injections.
  Case "method_self-method_self".
    eapply binds_fun; eassumption.
  Case "method_self-method_other".
    contradiction by (eauto using binds_nobinds).
  Case "method_other-method_self".
    contradiction by (eauto using binds_nobinds).
  Case "method_other-method_other".
    eapply tr_method_fun; eassumption.
Qed.
```

By the *sub_ty* rule, an object can be a supertype of only itself.

```
Fact ob_sub_ty_is_same_ob: ∀ t t' O,
    sub_ty t t' ∧ (t' = otyp2typ O) → (t = otyp2typ O).
Proof.
  intros t t' O H.
  inversion H; clear H.
  induction H0; subst; try discriminate; auto.
Qed.

Hint Resolve ob_sub_ty_is_same_ob.
```

Two generalized typing rules for fields:

```
Lemma gt_field: ∀ E e0 f t O,
    wide_typing E e0 (otyp2typ O) →
    field O f t →
    exp_ty E (e_field e0 f) t.
Proof.
  intros.
  inversion H; subst.
  assert (t0 = otyp2typ O) by eauto.
  subst; eauto.
Qed.

Hint Resolve gt_field.

Lemma gt_sub : ∀ E e t t',
    wide_typing E e t →
    sub_ty t t' →
    wide_typing E e t'.
```

```
Proof.
  destruct 1; eauto.
Qed.

Hint Extern 1 (wide_typing ?E ?e ?t) ⇒
match goal with
| H: sub_ty ?t0 t ⊢ _ ⇒ refine (gt_sub E e t0 t _ H)
end.
```

7.2.1 Properties from *ok_ot*, *ok_tt*, *ot_noobj*, and *tt_noobj*

```
Module Type OkTable.
    Parameter ot_noobj: Object \notin dom OT.
    Parameter tt_noobj: Object \notin dom TT.
    Parameter ok_ot: ok_otable OT.
    Parameter ok_tt: ok_ttable TT.
End OkTable.

Module OkTableFacts (Import H: OkTable).

    Definition ok_ot_class O T fs ms := ok_otable_class _ O T fs ms ok_ot.
    Definition ok_tt_class T T' ms := ok_ttable_class _ T T' ms ok_tt.
    Definition ok_ot_meth O T fs ms m E t e H :=
        ok_ob_class_meth O T fs ms m E t e (ok_ot_class _ _ _ _ H).
    Definition ok_tt_meth T T' ms m E t e H :=
        ok_tr_class_meth T T' ms m E t e (ok_tt_class _ _ _ H).

    Hint Resolve ok_ot_class ok_tt_class ok_ot_meth ok_tt_meth.
```

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

```
    Fact obj_nofields : ∀ flds, tr_fields Object flds → flds = nil.
    Proof.
      intros; inversion_clear H; subst; reflexivity.
    Qed.

    Fact obj_nometh : ∀ m mdecl, ~tr_method Object m mdecl.
    Proof.
      unfold not; intros.
      inversion_clear H; subst; apply tt_noobj; eapply binds_In; eassumption.
    Qed.

    Hint Resolve obj_nofields obj_nometh.

    Fact ok_def_ob_empty: ∀ O, def_ob_ty O Object nil nil.
    Proof.
      intros; unfold def_ob_ty; split; auto.
    Qed.

    Fact ok_def_tr_empty: ∀ T, def_tr_ty T Object nil.
    Proof.
      intros; unfold def_tr_ty; split; auto.
    Qed.

    Fact tr_fields_fun: ∀ T fs fs',
        tr_fields T fs → tr_fields T fs' → fs = fs'.
    Proof.
      intros T fs fs' H H'.
      inversion H; inversion H'; reflexivity.
    Qed.
```

```
Fact ob_fields_fun: ∀ O fs fs',
    ob_fields O fs → ob_fields O fs' → fs = fs'.
Proof.
  intros O fs fs' H H'. revert fs' H'.
  induction H; intros; inversion H'; subst.
  assert ((T, fs, ms) = (T0, fs', ms0))
    by (eapply binds_fun; eassumption).
  subst injections.
  reflexivity.
Qed.
```

Properties that follow from the *ok_ot* and *ok_tt* assumptions.

```
Fact tr_method_implies_typing: ∀ 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.
Proof.
  intros. remember (E, t0, e) as mdecl.
  induction H; subst.
  Case "tr_method_self".
    eauto via (meth_tr_ty T T' m E t0 e).
  Case "tr_method_other".
    destruct IHtr_method as (t1, H2, H3); eauto.
    assert (sub_ty (ttyp2typ T) t1) by eauto.
    ∃ t1; eauto.
Qed.
```

```
Fact ob_table_implies_subtyping : ∀ O T fs ms,
    binds O (T, fs, ms) OT → sub_ty (otyp2typ O) (ttyp2typ T).
Proof.
  intros.
  assert (ob_extends O T)
    by (unfold ob_extends; ∃ fs; ∃ ms; exact H).
  eapply s_ob_tapp; exact H0.
Qed.
```

```
Fact ob_method_implies_typing: ∀ O m E t0 e,
    ob_method O m (E, t0, e) →
    exists2 t', sub_ty (otyp2typ O) t' & wide_typing ((self,t')::E) e t0.
Proof.
  intros. remember (E,t0,e) as mdecl.
  destruct H; subst.
  Case "ob_method_self".
    eauto via (meth_ob_ty O T m E t0 e).
  Case "ob_method_other".
    remember (E, t0, e) as mdecl.
    inversion H1; subst.
    SCase "tr_method_self".
      eauto via (meth_tr_ty T T' m E t0 e).
      ∃ (ttyp2typ T); assert (sub_ty (otyp2typ O) (ttyp2typ T))
        by eauto; eauto.
    SCase "tr_method_super".
      assert (sub_ty (otyp2typ O) (ttyp2typ T)) by eauto; eauto.
      assert ((E, t0, e) = (E, t0, e)
        → exists2 t' : typ, sub_ty (ttyp2typ T) t'
```

```
      & wide_typing ((self, t')::E) e t0).
    intros.
    eauto using tr_method_implies_typing.
    destruct H6; [reflexivity | eauto].
Qed.

Hint Resolve tr_method_implies_typing ob_method_implies_typing.
```

Subtyping preserves method types.

```
Fact sub_tr_tr_mtype_aux : ∀ u1 t1 m ts t0 te u t,
    sub_ty u1 t1 →
    u1 = ttyp2typ u → t1 = ttyp2typ t →
    tr_method t m (ts, t0, te)→
    exists2 us, imgs ts = imgs us &
      exists2 t0', sub_ty t0' t0 &
        ∃ ue, tr_method u m (us, t0', ue).
Proof.
  intros u t m ts t0 te u' t' H. revert m ts t0 te u' t'.
  induction H; intros; try discriminate.
  Case "s_obj".
    subst; inversion H0; subst.
    assert (∀ m mdecl, ~tr_method Object m mdecl) by eauto.
    elim (H m (ts, t0, te)); assumption.
  Case "s_refl".
    subst; inversion H0; subst; eauto.
  Case "s_trans".
    destruct t2.
    SCase "t2 = ttyp t".
      destruct IHsub_ty2 with m ts t0 te t t' as (us2, IH2a, IH2b); eauto.
      destruct IH2b as (t02, IH2c, (ue2, IH2e)).
      destruct IHsub_ty1 with m us2 t02 ue2 u' t as (us1, IH1a, IH1b); eauto.
      destruct IH1b as (t01, IH1c, (ue1, IH1e)).
      ∃ us1; [ eauto using trans_eq | eauto ].
    SCase "t2 = otyp t".
      subst.
      assert (ttyp2typ u' = otyp2typ o) by eauto; inversion H1.
  Case "s_tr_tapp".
    inversion H0; clear H0.
    inversion H1; clear H1.
    subst.
    destruct H as (ms, H).
    case_eq (get m ms); [ intros ((us, t1),ue) Hb | intro Hb ].
    SCase "binds m ms (us, t1, ue)".
      assert (Hok: meth_tr_ty u' t' m us t1 ue) by eauto.
      destruct Hok as (Hoverride, _).
      destruct Hoverride with ts t0 te; eauto.
    SCase "no_binds m ms".
      eauto.
Qed.

Fact sub_tr_tr_mtype: ∀ u t m t0 ts te,
    sub_ty (ttyp2typ u) (ttyp2typ t) → tr_method t m (ts, t0, te) →
    exists2 us, imgs ts = imgs us &
      exists2 t0', sub_ty t0' t0 &
```

```
        ∃ ue, tr_method u m (us, t0', ue).
Proof.
  intros; eauto using sub_tr_tr_mtype_aux.
Qed.

Fact sub_tr_ob_mtype_aux : ∀ u1 t1 m ts t0 te u t,
    sub_ty u1 t1 →
    u1 = otyp2typ u → t1 = ttyp2typ t →
    tr_method t m (ts, t0, te)→
    exists2 us, imgs ts = imgs us &
      exists2 t0', sub_ty t0' t0 &
        ∃ ue, ob_method u m (us, t0', ue).
Proof.
  intros u t m ts t0 te u' t' H. revert m ts t0 te u' t'.
  induction H; intros; try discriminate.
  Case "s_obj".
    assert (~tr_method Object m (ts, t0, te)) by eauto.
    inversion H0; clear H0; subst; contradiction.
  Case "s_refl".
    subst; discriminate.
  Case "s_trans".
    destruct t2.
    SCase "t2 = ttyp2typ t".
      clear IHsub_ty2.
      assert (exists2 us, imgs ts = imgs us
        & exists2 t0', sub_ty t0' t0
        & ∃ ue, tr_method t m (us, t0', ue)).
          subst; eauto using sub_tr_tr_mtype.
      destruct H4 as (us, H4, (t0', H5, (ue, H6))).
      destruct IHsub_ty1
        with m us t0' ue u' t as (us', H7, (t0'', H8, (ue', H9))) ; eauto.
      assert (sub_ty t0'' t0) by eauto.
      assert (imgs ts = imgs us'). rewrite → H4; assumption.
      eauto.
    SCase "t2 = otyp2typ t".
      clear IHsub_ty1; subst.
      assert (otyp2typ u' = otyp2typ o) by eauto.
      inversion H1; clear H1; subst.
      eapply IHsub_ty2; eauto.
  Case "s_ob_tapp".
    subst.
    inversion H0; clear H0.
    inversion H1; clear H1.
    subst.
    destruct H as (fs, (ms, H)).
    case_eq (get m ms); [intros ((us, t1), ue) Hb | intro Hb].
    SCase "binds m ms (us, t1, te)".
      assert (Hok: meth_ob_ty u' t' m us t1 ue) by eauto.
      destruct Hok as (Hoverride, _).
      destruct Hoverride with ts t0 te; eauto.
    SCase "no_binds m ms".
      eauto.
Qed.
```

```
Fact sub_tr_ob_mtype: ∀ u t m t0 ts te,
    sub_ty (otyp2typ u) (ttyp2typ t) → tr_method t m (ts, t0, te) →
    exists2 us, imgs ts = imgs us &
      exists2 t0', sub_ty t0' t0 &
      ∃ ue, ob_method u m (us, t0', ue).
Proof.
  intros; eauto using sub_tr_ob_mtype_aux.
Qed.
```

Two generalized typing rules for methods:

```
Lemma gt_tr_meth : ∀ E E0 e0 b t0 t m es,
    wide_typing E e0 (ttyp2typ t0) →
    tr_method t0 m (E0, t, b) →
    wide_typings E es (imgs E0) →
    wide_typing E (e_method e0 m es) t.
Proof.
  intros.
  inversion H; subst.
  induction t1.
  Case "sub_type is trait type".
    destruct sub_tr_tr_mtype with (1:= H3) (2:= H0)
      as (us, Hu, (t0', Hs, (ue, Hm))).
    rewrite Hu in H1; eauto.
  Case "sub_type_is_object_type".
    destruct sub_tr_ob_mtype with (1:= H3) (2:= H0)
      as (us, Hu, (t0', Hs, (ue, Hm))).
    rewrite Hu in H1; eauto.
Qed.

Lemma gt_ob_meth : ∀ E E0 e0 b t0 t m es,
    wide_typing E e0 (otyp2typ t0) →
    ob_method t0 m (E0, t, b) →
    wide_typings E es (imgs E0) →
    wide_typing E (e_method e0 m es) t.
Proof.
  intros. inversion H; subst.
  assert (t1 = otyp2typ t0) by eauto.
  subst; eauto.
Qed.

Hint Resolve gt_tr_meth gt_ob_meth.

End OkTableFacts.
```

8 FBCF Properties

In this file, we prove two main theorems and two main lemmas for type soundness. First, we prove the *weakening* lemma and the *Term_substitutivity* lemma. Secondly, we prove the *preservation* theorem and the *progress* theorem.

```
Require Import AdditionalTactics.
Require Import Metatheory.
Require Import FBCF_Definition.
Require Import FBCF_Facts.
```

8.1 Weakening Lemma (Optional)

The *metatheory* library already guarantees this property. We prove this lemma to show that this property is provable. To handle a mutually inductive definition of expression typing, we prove the *weakening'* lemma before the *weakening* lemma.

```
Lemma weakening':
    (∀ EE e t, exp_ty EE e t →
      ∀ E F G, EE=F++G → ok (F++E++G) → exp_ty (F++E++G) e t) ∧
    (∀ EE e t, wide_typing EE e t →
      ∀ E F G, EE=F++G → ok (F++E++G) → wide_typing (F++E++G) e t) ∧
    (∀ EE es env, wide_typings EE es env →
      ∀ E F G, EE=F++G → ok (F++E++G) → wide_typings (F++E++G) es env).
Proof.
  apply typings_mutind; intros; subst; eauto using binds_weaken.
Qed.

Lemma weakening: ∀ E F G e t,
    ok (F++E++G) →
    exp_ty (F++G) e t →
    exp_ty (F++E++G) e t.
Proof.
  intros.
  destruct weakening' as (Hweak, _).
  eapply Hweak; trivial; assumption.
Qed.
```

8.2 Requirements for Preservation

8.2.1 Preservation over Evaluation Contexts

```
Fact preservation_over_esc: ∀ EE E ts e e',
    exps_context EE →
    (∀ t, exp_ty E e t → wide_typing E e' t) →
    wide_typings E (EE e) ts →
    wide_typings E (EE e') ts.
Proof.
  intros. generalize dependent ts.
  induction H; intros; inversion H1; subst; try eauto.
  Case "esc_head".
    destruct H6; eauto.
Qed.

Module PROPERTIES (H: HYPS).
```

Import previous conclusions for the hypotheses.

```
  Module Import FACTS := OKTABLEFACTS H.

  Fact preservation_over_ec: ∀ EE E e e' t,
      exp_context EE →
      (∀ (t0 : typ), exp_ty E e t0 → wide_typing E e' t0) →
      exp_ty E (EE e) t →
      wide_typing E (EE e') t.
  Proof.
    intros.
    destruct H; inversion H1; subst; eauto using preservation_over_esc.
```

```
Qed.
```

8.2.2 Term Substitutivity

To handle a mutually inductive definition of expression typing, we define the *Term_substitutivity'* lemma before the *Term_substitutivity* lemma.

```
Lemma term_substitutivity': ∀ E E0 ds Eds,
    wide_typings E0 ds (imgs E) →
    env_zip E ds Eds →
    (∀ EE e t, exp_ty EE e t →
        EE = E → wide_typing E0 (subst_exp Eds e) t) ∧
    (∀ EE e t, wide_typing EE e t →
        EE = E → wide_typing E0 (subst_exp Eds e) t) ∧
    (∀ EE es ts, wide_typings EE es ts →
        EE = E → wide_typings E0 (List.map (subst_exp Eds) es) ts).
Proof.
  intros E E0 ds Eds Hb Hz.
  apply typings_mutind; intros; subst; simpl; eauto.
  Case "t_var".
    destruct binds_zip with (1:=Hb) (2:=Hz) (3:=H0) as (e, H1a, H1b).
    rewrite H1a. exact H1b.
  Case "t_self".
    destruct binds_zip with (1:=Hb) (2:=Hz) (3:=H0) as (e, H1a, H1b).
    rewrite H1a. exact H1b.
Qed.

Lemma term_substitutivity: ∀ 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.
Proof.
  intros.
  destruct term_substitutivity' with (1:=H0) (2:=H1) as (_, (Hsub, _)).
  eapply Hsub; trivial; assumption.
Qed.
```

8.3 Preservation

```
Theorem preservation: ∀ 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 (O0 = O) 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.
```

8.4 Progress

To handle a mutually inductive definition of expression typing, we prove the *progress'* theorem before the *progress* theorem.

```
Theorem progress' :
    (∀ E e t, exp_ty E e t →
    E = nil → val e ∨ ∃ e', eval_rule e e')
    ∧
    (∀ E e t, wide_typing E e t →
    E = nil → val e ∨ ∃ e', eval_rule e e')
    ∧
    (∀ E ds env, wide_typings E ds env →
    E = nil → (∀ v, ln v ds → val v) ∨
    exists2 EE, exps_context EE & (exists2 e0, (EE e0) =
    ds & (∃ 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; ∃ (fun e1 ⇒ e1::es); eauto.
      SCase "es_progress".
        subst es; right; ∃ (fun e1 ⇒ e::(EE e1)); eauto.
  Qed.

  Theorem progress: ∀ e t,
      exp_ty nil e t →
      val e ∨ (∃ e', eval_rule e e').
  Proof.
    intros.
    destruct (progress') as (Hpro,_).
    apply Hpro with (1:=H); reflexivity.
  Qed.
End PROPERTIES.
```

8.5 Epilogue

Check that these properties indeed prove the type safety.

```
Module SAFETYPROOF : SAFETY := PROPERTIES.
```

References

[1] Cast-free featherweight java. http://soft.vub.ac.be/~bdefrain/featherj/.

[2] The coq proof assistant. http://coq.inria.fr/.

[3] The fortress programming language. http://projectfortress.sun.com/Projects/Community.

[4] Poplmark wiki. http://alliance.seas.upenn.edu/~plclub/cgi-bin/poplmark.

[5] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification Version 1.0, March 2008.