Coq Mechanizations @ KAIST

ERC Workshop (25 June to 28 June, 2011) Jieung Kim (KAIST) Sunggyeong Bae (KAIST) Sukyoung Ryu (KAIST)

FFMM Mechanization

Motivation

- Defined a formal calculus that has symmetric multiple dispatch and symmetric multiple inheritance
- Mechanize the calculus and its type safety using a proof assistant tool, Coq

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.

FFMM

Fortress is a programming language for scientists and engineers. **F**eatherweight **F**ortress with ***M**ultiple Dispatch* and ***M**ultiple Inheritance* (FFMM) illustrates a core calculus for Fortress, which has multiple dispatch and multiple inheritance.

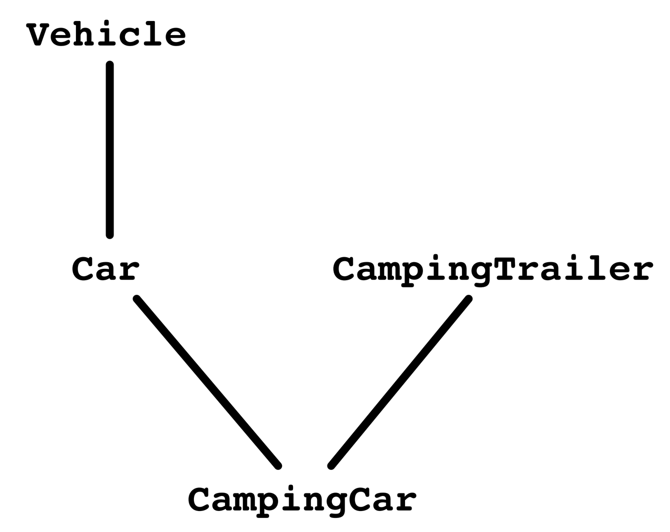
- **Multiple dispatch**: Allows method selection among overloaded methods at run time based on dynamic types of more than one method arguments
- **Multiple inheritance**: Allows a type to have more than one super type

Syntax of FFMM

T	trait name	O	object name
m	method name	f	field name
x	method parameter name		

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

Ambiguity Problems




```
collide(Car c, CampingCar cc)
collide(CampingCar cc, Car c)
```

```
LightOn(Car c)
LightOn(CampingTrailer ct)
```

```
tow(Vehicle v, Car c)
tow(Car c, Vehicle v)
tow(CampingCar cc1, CampingCar(cc2)
```

Have to check overloaded method declarations statically to guarantee no ambiguous calls at run time

Overloading Rules

Informal Description

The rules determine whether a set of overloaded declarations is valid by considering every pair of the declarations in the set independently. **A pair** is valid if **it satisfies one of the following rules**:

Exclusion Rule

If the parameter types of the declarations are **disjoint types**, then the pair is a valid overloading.

```
collide(Car c, Car c)
collide(Car c, Car c, Car c)
```

Subtype Rule

If the parameter type of one declaration is **a strict subtype** of the parameter type of the other declaration, and the return type of the former is a subtype of the return type of the latter, then the pair is a valid overloading.

```
LightOn(Car c)
LightOn(CampingCar cc)
```

Meet Rule

If the parameter types of the declaration are **not in the subtype relation**, then the pair is a valid overloading if there is a declaration whose parameter type is **an intersection type** of the parameter types of the declarations.

```
tow(Vehicle v, Car c)
tow(Car c, Vehicle v)
tow(Car c, Car c)
```

Modular Multiple Dispatch with Multiple Inheritance. *(In Proceedings of the 2007 ACM symposium on applied computing)*

Formal Calculus

[ValidMeth]
$$\frac{\begin{array}{c}\forall\{(md, C), (md', C')\} \subseteq visible_p(C^o).\\ md \neq md', \ \text{(not same declaration)}\\ md = m(\overrightarrow{\_ : \tau^a}) : \tau^r \_, \qquad md' = m(\overrightarrow{\_ : \tau^{a'}}) : \tau^{r'} \_,\\ p \vdash valid(m, C, \overrightarrow{\tau^a} \to \tau^r, C', \overrightarrow{\tau^{a'}} \to \tau^{r'}, visible_p(C^o))\end{array}}{p \vdash validMeth(C^o)}$$

[ValidExc]
$$\frac{|\overrightarrow{\tau^a}| \neq |\overrightarrow{\tau^{a'}}|}{p \vdash valid(m, C, \overrightarrow{\tau^a} \to \tau^r, C', \overrightarrow{\tau^{a'}} \to \tau^{r'}, S)}$$

[ValidSubTyR]
$$\frac{\begin{array}{c}|\overrightarrow{\tau^a}| = |\overrightarrow{\tau^{a'}}| \qquad C\ \overrightarrow{\tau^a} \neq C'\ \overrightarrow{\tau^{a'}} \qquad p \vdash \overrightarrow{\tau^{a'}} <: \overrightarrow{\tau^a}\\ p \vdash \tau^{r'} <: \tau^r \qquad p \vdash C' <: C\end{array}}{p \vdash valid(m, C, \overrightarrow{\tau^a} \to \tau^r, C', \overrightarrow{\tau^{a'}} \to \tau^{r'}, S)}$$

[ValidSubTyL]
$$\frac{\begin{array}{c}|\overrightarrow{\tau^a}| = |\overrightarrow{\tau^{a'}}| \qquad C\ \overrightarrow{\tau^a} \neq C'\ \overrightarrow{\tau^{a'}} \qquad p \vdash \overrightarrow{\tau^a} <: \overrightarrow{\tau^{a'}}\\ p \vdash \tau^r <: \tau^{r'} \qquad p \vdash C <: C'\end{array}}{p \vdash valid(m, C, \overrightarrow{\tau^a} \to \tau^r, C', \overrightarrow{\tau^{a'}} \to \tau^{r'}, S)}$$

[ValidMeet]
$$\frac{\begin{array}{c}l = |\overrightarrow{\tau^a}| = |\overrightarrow{\tau^{a'}}| \qquad C\ \overrightarrow{\tau^a} \neq C'\ \overrightarrow{\tau^{a'}} \qquad \tau_0^a = C \qquad \tau_0^{a'} = C'\\ \exists (m(\overrightarrow{\_ : \tau^{a''}}) : \_, \tau_0^{a''}) \in S.\\ \text{where } (l = |\overrightarrow{\tau^{a''}}|) \wedge (\forall\, 0 \leq i \leq l.\, p \vdash meet(\{\tau_i^a,\ \tau_i^{a'},\ \tau_i^{a''}\}, \tau_i^{a''}))\end{array}}{p \vdash valid(m, C, \overrightarrow{\tau^a} \to \tau^r, C', \overrightarrow{\tau^{a'}} \to \tau^{r'}, S)}$$

Coq Implementation

```
Definition validmeet' (mn: mname) (tys : list typ) (ty : typ) (mS : mSet) : Prop :=
    exists2 mdt, mdt \in mS &
        ((tys = (getartys mdt)) ∧ (ty = (snd mdt)) ∧ (mn = (getmname mdt))).
Inductive valid (mdt1 mdt2 : mdttype) (mS : mSet) : Prop :=
  | valid_same :
      (getmid mdt1) = (getmid mdt2) →
      (snd mdt1) = (snd mdt2) →
      valid mdt1 mdt2 mS
  | valid_diff_name :
      (getmname mdt1) ≠ (getmname mdt2) →
      valid mdt1 mdt2 mS
  | valid_exc :
      (getmid mdt1) ≠ (getmid mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
      (getmname mdt1) = (getmname mdt2) →
      (getenvlen mdt1) ≠ (getenvlen mdt2) →
      valid mdt1 mdt2 mS
  | valid_sub_ty_r :
      (getmid mdt1) ≠ (getmid mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
      (getartys mdt1) ≠ (getartys mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
      (getmname mdt1) = (getmname mdt2) →
      (getenvlen mdt1) = (getenvlen mdt2) →
      sub_tys (getartys mdt2) (getartys mdt1) →
      sub_ty (getrty mdt2) (getrty mdt1) →
      sub_ty (snd mdt2) (snd mdt1) →
      valid mdt1 mdt2 mS
  | valid_sub_ty_l :
      (getmid mdt1) ≠ (getmid mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
      (getartys mdt1) ≠ (getartys mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
      (getmname mdt1) = (getmname mdt2) →
      (getenvlen mdt1) = (getenvlen mdt2) →
      sub_tys (getartys mdt1) (getartys mdt2) →
      sub_ty (getrty mdt1) (getrty mdt2) →
      sub_ty (snd mdt1) (snd mdt2) →
      valid mdt1 mdt2 mS
  | valid_meet : ∀ tys ty,
      (getmid mdt1) ≠ (getmid mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
      (getartys mdt1) ≠ (getartys mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
      (getmname mdt1) = (getmname mdt2) →
      (getenvlen mdt1) = (getenvlen mdt2) →
      ~ (sub_tys (getartys mdt1) (getartys mdt2)) →
      ~ (sub_tys (getartys mdt2) (getartys mdt1)) →
      is_tys (getartys mdt1) (getartys mdt2) tys →
      is_ty (snd mdt1) (snd mdt2) ty →
      validmeet' (getmname mdt1) tys ty mS→
      valid mdt1 mdt2 mS.
```

Bigraph Library in Coq

Motivation

- Represent bipartite graphs using Coq
- Mechanize several theorems in Graph Theory

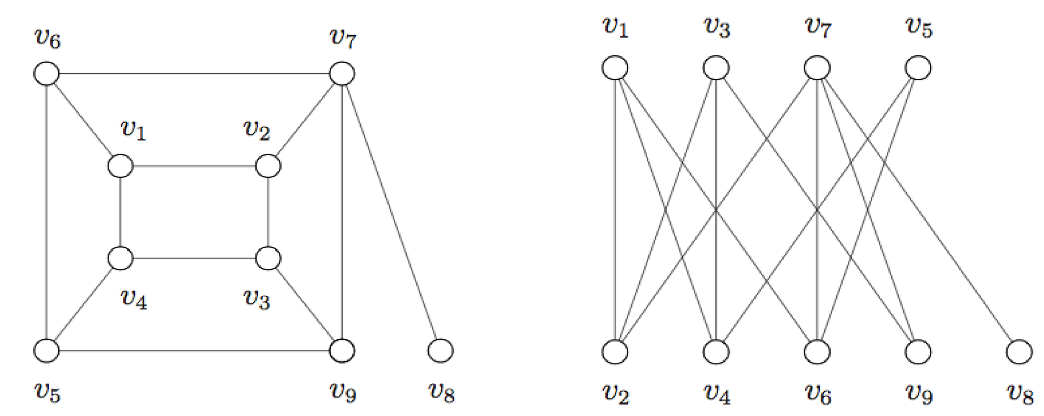
Bipartite Graph

Mathematical Representation

- **Definition**

A graph G is called *bipartite* if it is possible to partition the vertex set of G into two subsets, say V_1 and V_2, so that every edge of G joins a vertex of V_1 with a vertex of V_2, and no vertex joins another vertex of its own set.

- **Example**



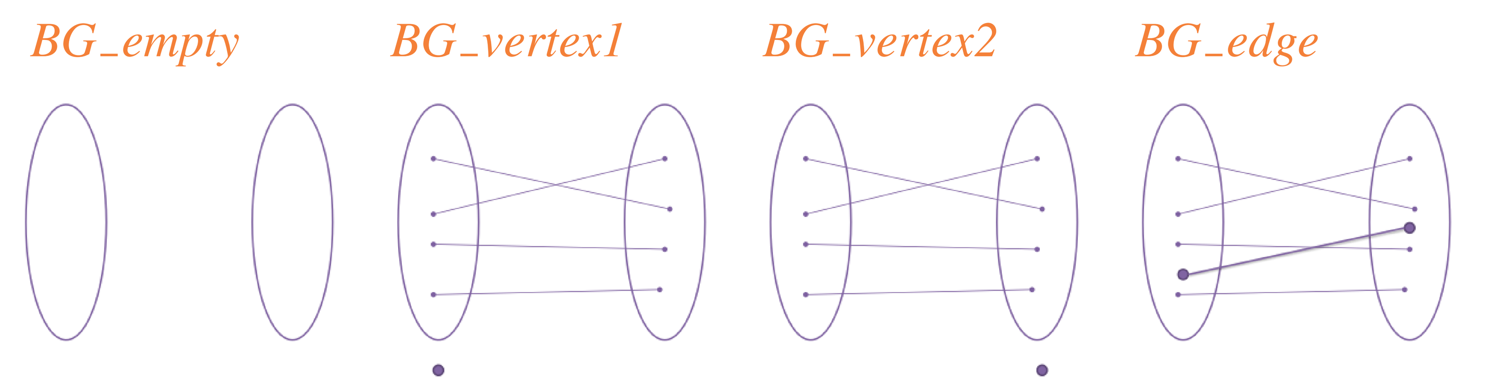
Inductive Definition

We can construct a **bigraph** with two sets of vertices V_1 and V_2 and a set of edges A using one of the following rules:

1. An empty **bigraph** is constructed with two empty sets of vertices and an empty set of edges.
2. With a **bigraph** upon V_1, V_2 and A, by adding a new vertex x ($\notin V_1$) to V_1, a **bigraph** upon $V_1 \cup \{x\}$, V_2 and A is constructed.
3. With a **bigraph** upon V_1, V_2 and A, by adding a new vertex y ($\notin V_2$) to V_2, a **bigraph** upon V_1, $V_2 \cup \{y\}$ and A is constructed.
4. With a **bigraph** upon V_1, V_2 and A, by adding a new edge xy such that $x \in V_1$, $y \in V_2$ and $\{xy\} \cap A = \emptyset$, a **bigraph** upon V_1, V_2 and $A \cup \{xy\}$ is constructed.

Coq Implementation

```
Inductive Bigraph : V_set → V_set → A_set → Type :=
  | BG_empty : Bigraph V_empty V_empty A_empty
  | BG_vertex1 :
     ∀ (v1 v2 : V_set) (a : A_set) (d : Bigraph v1 v2 a) (x : Vertex),
       ¬ v1 x → ¬ v2 x →
       Bigraph (V_union (V_single x) v1) v2 a
  | BG_vertex2 :
     ∀ (v1 v2 : V_set) (a : A_set) (d : Bigraph v1 v2 a) (y : Vertex),
       ¬ v1 y → ¬ v2 y →
       Bigraph v1 (V_union (V_single y) v2) a
  | BG_edge :
     ∀ (v1 v2 : V_set) (a : A_set) (d : Bigraph v1 v2 a) (x y : Vertex),
       v1 x → v2 y → x ≠ y →
       ¬ a (A_ends x y) → ¬ a (A_ends y x) →
       Bigraph v1 v2 (A_union (E_set x y) a).
```



What Is Next?

- Mechanize proofs of the following theorems:

König's theorem: In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.

Equivalencies of König's theorem with seven theorems: the Menger's Theorem, the König's theorem for matrices, the König-Egerváry theorem, the Hall's marriage theorem, the Birkhoff-Von Neumann theorem, the Dilworth's theorem, and the Max Flow-Min Cut theorem.

- Implement a new Graph library

For compatibility with the Sets library in the Coq standard library