IFIP WG2.8

Project Fortress: from SunLabs to KAIST

or, from Industrial Labs to Academia

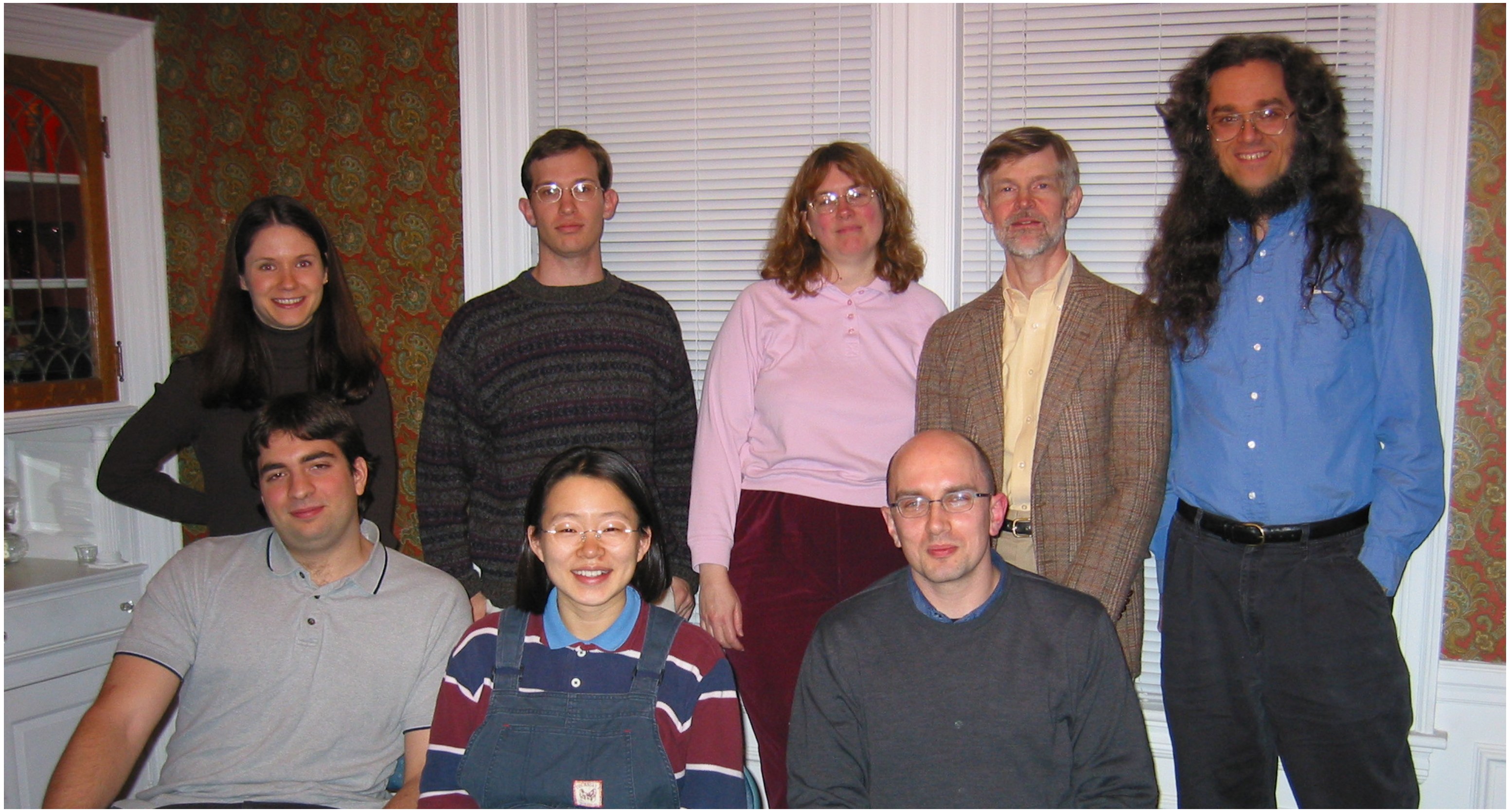
Sukyoung Ryu

Department of Computer Science
Korea Advanced Institute of Science and Technology

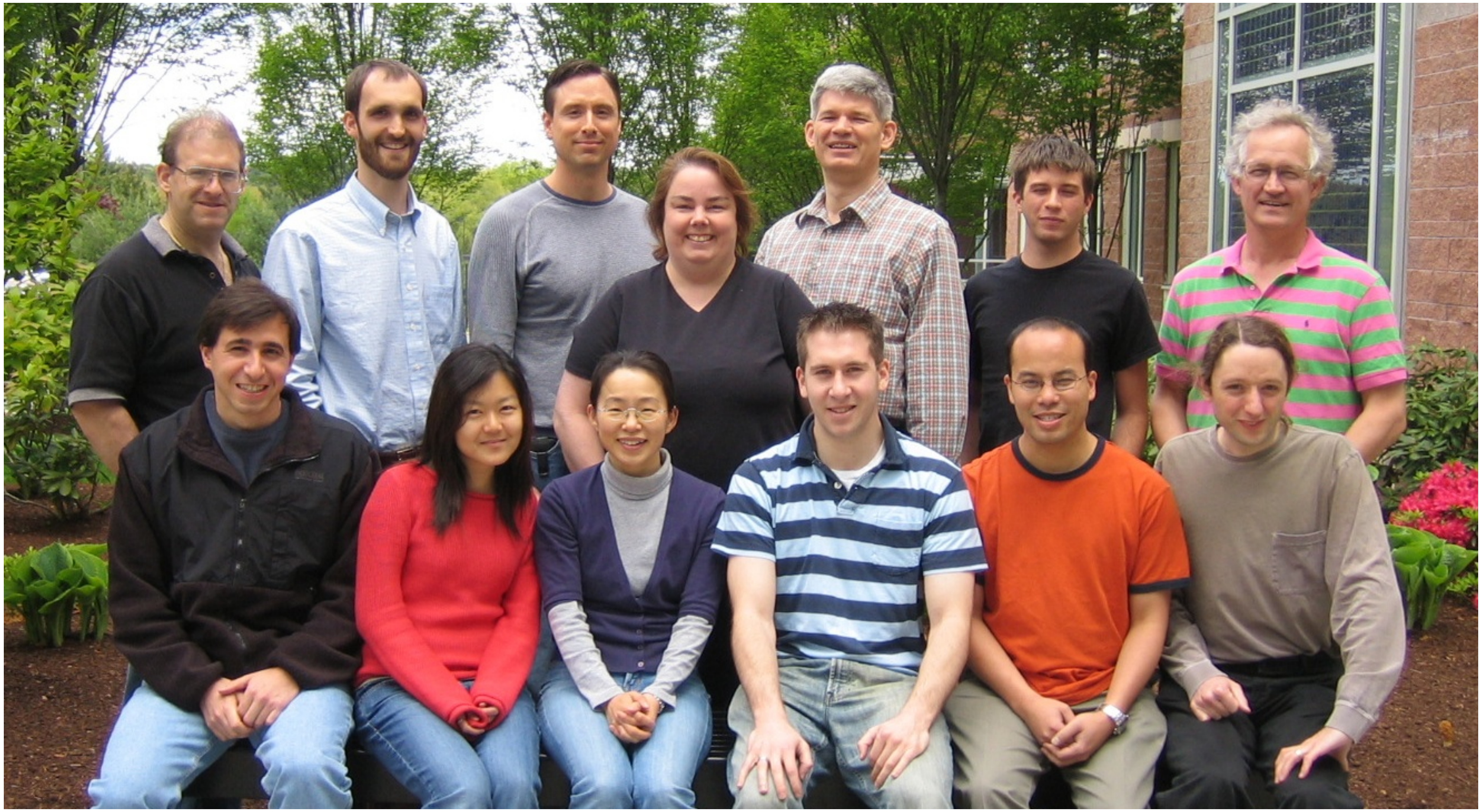
April 13, 2010

Static Program Analysis
(at KAIST)

Debugging Everywhere
(at Harvard)



Fortress Programming Language
(at Sun Labs.)



KAIST Computer Science

Project Fortress

- A multicore language for scientists and engineers
- Run your whiteboard in parallel!

$$v_{\mathrm{norm}} = \underline{v/\underline{\|v\|}}$$

$$\sum_{\underline{k \leftarrow 1:n}} \underline{a_k}\ \underline{x^k}$$

$$C = \underline{A \cup B}$$

$$y = \underline{\underline{3x}\ \underline{\sin x}}\ \underline{\cos \underline{2x}}\ \underline{\log \underline{\log x}}$$

- "Growing a Language"

  Guy L. Steele Jr., keynote talk, OOPSLA 1998

Project Fortress: History

- Fortress is a growable, mathematically oriented, parallel programming language for scientific applications.
- Started under Sun/DARPA HPCS program, 2003–2006.
- Fortress is now an open-source project with international participation.
- The Fortress 1.0 release (March 2008) synchronized the specification and implementation.
- Moving forward, we are growing the language and libraries and developing a compiler.

Project Fortress: Sales Pitch

- Convolver in Satnam Singh's slides yesterday

```
for (int i = 0; i < a.Length; i++)
    ypar += a[i] * A.Shift(xpar, -i);
```

- Convolver in Fortress

$$y_t = \sum_{k \leftarrow 0 \# N} a_k \, x_{t-k}$$

- "Birdcount" programs[a]
  - > Collaboration with Mike Zody at the Broad Institute [b]
  - > Find chicken mutants with reference chicken genome

[a] http://projectfortress.sun.com/Projects/Community/browser/trunk/ProjectFortress/demos

[b] "Birds of a feather inherit together: Chicken breeds shed light on genes underlying domestic traits", http://www.broadinstitute.org/news/1430

KAIST Computer Science

Formalism for the Fortress Programming Language

Eric Allen
Eric.Allen@sun.com

Sukyoung Ryu
Sukyoung.Ryu@sun.com

Joe Hallett
Joseph.Hallett@sun.com

The Value of Formal Methods

Ariane 5

A data conversion from 64-bit floating point to 16-bit signed integer value raised an uncaught Overflow exception.

Result: The launcher was destroyed 40 seconds into the flight. The launch cost of an Ariane 5 was $180 million.

Mars Climate Orbiter

Orbiter software represented Force Time in Ns. Ground software represented Force Time in lbf s.

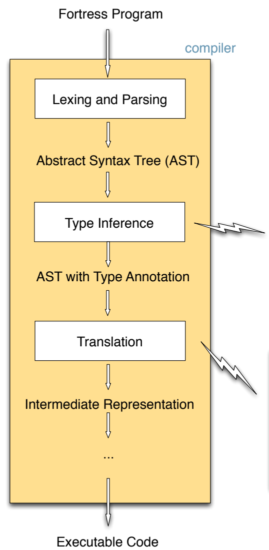
Result: The spacecraft was lost. The project cost was $327.6 million for both orbiter and lander.

Patriot Missile Failure

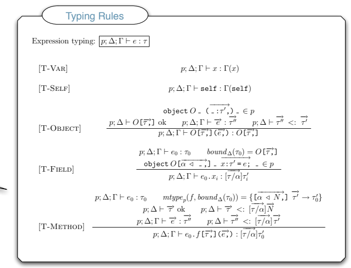
Accumulated rounding error in patriot missile software caused a missile to track its target incorrectly.

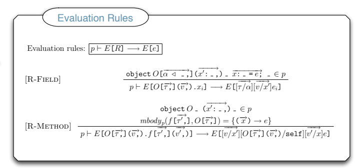
Result: SCUD missile was able to strike an army barrack, resulting in 28 Americans killed.

Formalized Semantics

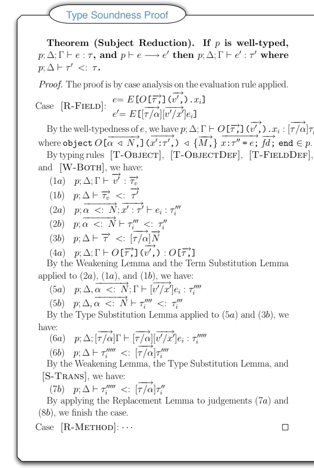


- Provides unambiguous specification for compiler writers
  - Fewer insidious bugs
  - More portable code





- Allows proofs of soundness and formal analysis



Example Program in Fortress

```
object Main[]() traits {Object}
  myself:Main[] = self
  identity[](x:Object):Object = x
end

Main[]().identity[](Main[]().myself)
```

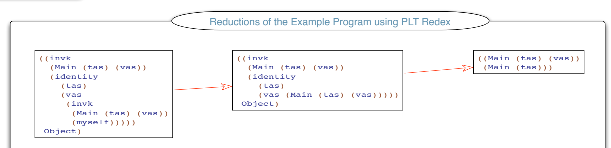
Mechanized Semantics

- Tests soundness of language semantics

Soundness of the Example Program

Suppose p is the example program.

If $p;\emptyset;\emptyset \vdash$ `Main[]().identity[](Main[]().myself)` : `Object`
and $p \vdash$ `Main[]().identity[](Main[]().myself)` $\longrightarrow^*$ `Main[]()`
then $p;\emptyset;\emptyset \vdash$ `Main[]()` : `Main[]` where $p;\emptyset \vdash$ `Main[]` $<:$ `Object`.



Formalism for Fortress

- Fortress calculi
  - Basic core Fortress
  - Core Fortress with where clauses
  - Core Fortress with overloading
  - Acyclic core Fortress with field definitions
- For each Fortress calculus
  - Syntax
  - Static semantics
  - Dynamic semantics
  - Type soundness proof

Core Fortress with Where Clauses

- "Hidden Type Variables and Conditional Extension for More Expressive Generic Programs," Joseph J. Hallett, Ph.D. Dissertation, Boston University, 2007

- "Implementing Hidden Type Variables in Fortress," Joe Hallett, Eric Allen, and Sukyoung Ryu. Chapter in book: *Semantic Engineering with PLT Redex*, Matthias Felleisen, Robby Findler, and Matthew Flatt. MIT Press. July 2009.

KAIST Computer Science

Programming Language Research Group

Programming Language Research Group

Home

- Members
  - Sukyoung Ryu
  - Jieung Kim
- Research
  - Publications
  - Talks
  - Seminars
  - Software
- Lectures

Programming Language Research Group at KAIST

Welcome to the Programming Language Research Group at KAIST. Our research interests include programming languages, compilers, program analyses, and programming environments. Our goal is to apply programming language technology to help programmers develop high-quality softwares. We are especially interested in applying advanced type systems, certifying compilers, automated and scalable program analyses, and practical debugging and testing tools to build safe and reliable systems.

We are actively recruiting MS/PhD students and postdoctoral researchers.

Department of Computer Science
Korea Advanced Institute of Science and Technology
Guseong-dong, Yuseong-gu
Daejeon 305-701, Korea

Programming Language Research Group

- Home
- Members
  - Sukyoung Ryu
  - Jieung Kim
- Research
  - Publications
  - Talks
  - Seminars
  - Software
- Lectures

류석영 Sukyoung Ryu

Assistant Professor
Computer Science Department
Korea Advanced Institute of Science and Technology

Research Interests

- Fortress, a multicore programming language for scientists and engineers
- Proof assistants for programming languages and program analyses
- Scalable, accurate, and practical program analyses
- Programming environments for debugging and testing

Education

PhD in Computer Science, September 1996 - August 2001
Computer Science Department, KAIST

MS in Computer Science, March 1995 - August 1996
Computer Science Department, KAIST

BS in Computer Science, March 1991 - February 1995
Computer Science Department, KAIST

Fortress Type System

- Traits are like Java™ interfaces, but may contain code
- Objects are like Java™ classes, but may not be extended
- Multiple inheritance of code (but not fields)
  - Objects with fields are the leaves of the hierarchy
- Traits and objects may be parameterized
  - Parameters may be types or compile-time constants
- Primitive types are first-class
  - Booleans, integers, floats, characters are all objects

Basic Core Fortress (BCF)

α, β			type variables
τ, τ', τ''	$::=$	$\alpha \quad \mid \quad \sigma$	type
σ	$::=$	$N \quad \mid \quad O[\![\overrightarrow{\tau}]\!]$	named type
N, M, L	$::=$	$T[\![\overrightarrow{\tau}]\!] \quad \mid \quad \mathrm{Object}$	trait type
p	$::=$	$\overrightarrow{d}\ e$	program
d	$::=$	$td \quad \mid \quad od$	definition
td	$::=$	$\mathtt{trait}\ T[\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!]\ \mathtt{extends}\ \{\overrightarrow{N}\}\ \overrightarrow{fd}\ \mathtt{end}$	trait definition
od	$::=$	$\mathtt{object}\ O[\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!](\overrightarrow{x{:}\tau})\ \mathtt{extends}\ \{\overrightarrow{N}\}\ \overrightarrow{fd}\ \mathtt{end}$	object definition
fd	$::=$	$f[\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!](\overrightarrow{x{:}\tau}){:}\tau = e$	method definition
e	$::=$	x	expression
	$\mid$	$\mathtt{self}$	
	$\mid$	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
	$\mid$	$e.x$	
	$\mid$	$e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	

BCF: Multiple Inheritance

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

KAIST Computer Science

BCF: Static Semantics

Method type lookup: $mtype_p(f, \tau) = \{[\![\overrightarrow{\alpha \texttt{ extends } N}]\!]\ \overrightarrow{\tau} \rightarrow \tau\}$

[MT-SELF]

$$\frac{\_\ C[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!]\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f[\![\overrightarrow{\beta \texttt{ extends } M}]\!](\overrightarrow{\_{:}\tau'}){:}\tau_0' = e \in \{\overrightarrow{fd}\}}{mtype_p(f, C[\![\overrightarrow{\tau}]\!]) = \{[\overrightarrow{\tau}/\overrightarrow{\alpha}][\![\overrightarrow{\beta \texttt{ extends } M}]\!]\ \overrightarrow{\tau'} \rightarrow \tau_0'\}}$$

[MT-SUPER]

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

[MT-OBJ]

$$mtype_p(f, \text{Object}) = \emptyset$$

BCF: Multiple Inheritance

Method type lookup: $mtype_p(f, \tau) = \{[\![\overrightarrow{\alpha \texttt{ extends } N}]\!]\ \overrightarrow{\tau} \to \tau\}$

[Mt-Self]
$$\frac{\_\ C[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!]\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f[\![\overrightarrow{\beta \texttt{ extends } M}]\!](\overrightarrow{\_:\tau'}):\tau'_0 = e \in \{\overrightarrow{fd}\}}{mtype_p(f, C[\![\overrightarrow{\tau}]\!]) = \{[\overrightarrow{\tau}/\overrightarrow{\alpha}][\![\overrightarrow{\beta \texttt{ extends } M}]\!]\ \overrightarrow{\tau'} \to \tau'_0\}}$$

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

[Mt-Obj]
$$mtype_p(f, \text{Object}) = \emptyset$$

BCF: Dynamic Semantics

v	$::=$	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})$	value
E	$::=$	$\Box$	evaluation context
	$\mid$	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e})$	
	$\mid$	$E.x$	
	$\mid$	$E.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
	$\mid$	$e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e})$	
R	$::=$	$v.x$	redex
	$\mid$	$v.f[\![\overrightarrow{\tau}]\!](\overrightarrow{v})$	

BCF: Nondeterminism

$$
\begin{array}{llll}
v & ::= & O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}) & \text{value} \\
E & ::= & \Box & \text{evaluation context} \\
 & | & O[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e}) & \\
 & | & E.x & \\
 & | & E.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) & \\
 & | & e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e}) & \\
R & ::= & v.x & \text{redex} \\
 & | & v.f[\![\overrightarrow{\tau}]\!](\overrightarrow{v}) &
\end{array}
$$

BCF in Coq: Multiple Inheritance

- With primitive recursion

```
(* Method type lookup
 *   mtype_p(f, sigma) = {[\ \overline{alpha extends N} \]\overline{ty} -> ty}
 *)
Definition mtype (p:P) (mn:m) (t:ty) : (list tv * list nty * list ty * ty) :=
  match t with
  | nty2ty(tty2nty tty_object) => (nil, nil, nil, ty_object)(* Mt-Obj *)
  | nty2ty(tty2nty (tty_tty tn tys)) => mtype' p mn (tcl2cl tn) tys(* trait *)
  ...                                                              (* object *)

Fixpoint mtype' (p:P) (mn:m) (name:cl) (tys:list ty)
                : (list tv * list nty * list ty * ty) :=
  let namedt := ...            (* convert cl to nty *)
  let ps := paths p namedt in  (* collect all the paths from namedt to Object *)
  let collected :=             (* collect all the methods from the paths *)
      filter (fun res => match res with Some sig => true | _ => false end)
             (map (fun (path:list nty) => mtype'' p mn path name tys) ps) in
  ...                          (* check there is only one and return it *)
```

KAIST Computer Science

BCF in Coq: Multiple Inheritance

```
(* Collect all the defined and inherited methods from a given path *)
Fixpoint mtype'' (p:P) (mn:m) (path :list nty) (name:cl) (tys:list ty)
                   : option (list tv * list nty * list ty * ty) :=
  match path with
  | nil => None
  | cons nt path' =>
    match get_decl p name with
    | Some decl =>
      match (find (fun d:md => match d with
                     md_def (ms_def mn' _ _ _) _ =>
                     if eq_nat_dec mn mn' then true else false
                   end) (cld2mds decl)) with
      | Some (md_def (ms_def _ tvds vds retty) _) => (* Mt-Self *)
        ...
      | None => (* Mt-Super *)
        mtype'' p mn path' name tys
      end
    ...
```

BCF in Coq: Multiple Inheritance

```
(* Collect all the paths from nt to Object *)
Definition paths (p:P) (nt:nty) : list (list nty) :=
  paths' p nt (length (get_decls p)).

Function paths' (p:P) (nt:nty) (bound:nat) {struct bound} : list (list nty) :=
  match bound with
  | S bound' =>
    let (tname,tas) := nty2nameTas nt in
    match get_decl p tname with
    | Some decl =>
      let sub := make_subst_tty tas (cld2tvs decl) in
      let supers := map sub (cld2supers decl) in
      fold_right (fun (sup:tty) (ps:list (list nty)) =>
                      (map (fun (l:list nty) => nt :: l)
                           (paths' p (tty2nty sup) bound')) ++ ps
                 ) nil supers
    | _ => nil (* !!! decl is not found; should be Object !!! *)
    end
  | _ => nil (* !!! bound not met !!! *)
  end.
```

BCF in Coq: Work in Progress

- Nondeterministic dynamic semantics
- Coercion between language constructs
- Test-driven development
  - Fortress $\longrightarrow$ BCF parser
  - Test programs
- Raising an exception vs static manipulation
  - Library Coq.Lists.List

```
Definition head (l:list) :=
  match l with
    | nil => error
    | x :: _ => value x
  end.
```

```
Definition hd (default:A) (l:list) :=
  match l with
    | nil => default
    | x :: _ => x
  end.
```

Core Fortress with Overloading

- Basic core Fortress (BCF) + overloading
- Overloading
  - > Multiple declarations for the same functional name visible in a single scope
  - > Several of the overloaded declarations may be applicable to any particular functional call

Functionals in Fortress

- Functionals
  - Functions
    - Top-level functions
    - Local functions
  - Methods
    - Dotted methods
    - Functional methods
- Special functionals
  - Operators
  - Coercions

Functionals in Fortress

- Functionals
  - Functions — first-class values
    - Top-level functions — top-level in components or APIs
    - Local functions — within blocks
  - Methods — have owners (traits or objects)
    - Dotted methods — implicit `self`
    - Functional methods — explicit `self`
- Special functionals
  - Operators — top-level functions or functional methods
  - Coercions — special dotted methods

Methods

- Methods are declared within traits or objects.
  - top-level in enclosing traits or objects
  - `self` is declared as a parameter of a method
- Dotted methods
  - invoked by a method call syntax
  - its receiver is bound to the `self` parameter
  - the value of `self` is the receiver
- Functional methods
  - invoked by a function call syntax
  - the corresponding argument is bound to the `self` param.
  - the value of `self` is the argument passed to it

Dotted Methods vs Functional Methods

$\texttt{trait}$ SequentialGenerator⟦E⟧ $\texttt{extends}$ { Generator⟦E⟧ }

  $seq(\texttt{self})$: SequentialGenerator⟦E⟧ $=$ $\texttt{self}$

  map⟦G⟧$(f\colon E \rightarrow G)$: SequentialGenerator⟦$G$⟧ $=$

    SimpleMappedSeqGenerator⟦E, G⟧$(\texttt{self}, f)$

  $\ldots$

$\texttt{end}$ SequentialGenerator

- Dotted methods: $g.map$⟦R⟧(f)
- Functional methods: $seq(g)$

Why Dotted Methods?

- Good for data extensibility

```
trait Flower
    color(): String
end

object Rose extends Flower
    color() = "Red"
end

object Lily extends Flower
    color() = "White"
end
```

KAIST Computer Science

Why Functions?

- For function extensibility with overloaded functions
  - > Multiple declarations with the same name

$$\mathit{color}(r\colon \mathrm{Rose}) = \text{“}\texttt{Red}\text{”}$$

$$\mathit{color}(r\colon \mathrm{Lily}) = \text{“}\texttt{White}\text{”}$$

  - > Dynamic dispatch selects the most specific definition at run time

$$\mathit{countRoses}(x\colon \mathrm{Flower}, y\colon \mathrm{Flower}) = 0$$

$$\mathit{countRoses}(x\colon \mathrm{Flower}, y\colon \mathrm{Rose}) = 1$$

$$\mathit{rose}\colon \mathrm{Flower} = \mathrm{Rose}$$

$$\mathit{countRoses}(\mathit{rose}, \mathit{rose})$$

Why Functional Methods?

- For data extensibility and encapsulation
- For function extensibility even with top-level functions
- For mathematical syntax with overloaded operators

```
trait Matrix excludes Vector
    opr ·(self, other: Vector): Matrix
    opr ·(other: Vector, self): Matrix
end
```

$v \cdot M + M \cdot v$

Fortress Overloading

- Goal: No ambiguous nor undefined calls at run time
- Challenges: Modular Multiple dispatch & Multiple inheritance[a]

[a] "Modular Multiple Dispatch with Multiple Inheritance," Eric Allen, J.J. Hallett, Victor Luchangco, Sukyoung Ryu, and Guy L. Steele Jr. SAC 2007: 22nd Annual ACM Symposium on Applied Computing

Fortress Overloading

- Goal: No ambiguous nor undefined calls at run time
- Challenges: Modular Multiple dispatch & Multiple inheritance[a]
  - > Multiple dispatch and ambiguity

$$countRoses(x\colon \text{Flower}, y\colon \text{Flower}) = 0$$
$$countRoses(x\colon \text{Flower}, y\colon \text{Rose}) = 1$$
$$countRoses(x\colon \text{Rose}, y\colon \text{Flower}) = 1$$

$$rose\colon \text{Flower} = \text{Rose}$$
$$countRoses(rose, rose) \qquad \text{(* Ambiguous call! *)}$$

[a] "Modular Multiple Dispatch with Multiple Inheritance," Eric Allen, J.J. Hallett, Victor Luchangco, Sukyoung Ryu, and Guy L. Steele Jr. SAC 2007: 22nd Annual ACM Symposium on Applied Computing

Fortress Overloading

- Goal: No ambiguous nor undefined calls at run time
- Challenges: Modular Multiple dispatch & Multiple inheritance[a]
  - > Multiple dispatch and ambiguity
  - > Multiple inheritance and ambiguity

```
trait Flower end
trait Thorny end
object Rose extends { Flower, Thorny } end
toString(x: Flower) = "Flower"
toString(x: Thorny) = "Thorny"
toString(Rose)            (* Ambiguous call! *)
```

[a] "Modular Multiple Dispatch with Multiple Inheritance," Eric Allen, J.J. Hallett, Victor Luchangco, Sukyoung Ryu, and Guy L. Steele Jr. SAC 2007: 22nd Annual ACM Symposium on Applied Computing

Fortress Overloading

- Goal: No ambiguous nor undefined calls at run time
- Challenges: Modular Multiple dispatch & Multiple inheritance[a]
  - > Multiple dispatch and ambiguity
  - > Multiple inheritance and ambiguity
  - > Modular check for ambiguity

[a] "Modular Multiple Dispatch with Multiple Inheritance," Eric Allen, J.J. Hallett, Victor Luchangco, Sukyoung Ryu, and Guy L. Steele Jr. SAC 2007: 22nd Annual ACM Symposium on Applied Computing

Fortress Overloading

- Goal: No ambiguous nor undefined calls at run time
- Challenges: Modular Multiple dispatch & Multiple inheritance[a]
  - Multiple dispatch and ambiguity
  - Multiple inheritance and ambiguity
  - Modular check for ambiguity
- Solution: Static overloading rules to guarantee the goal

[a] "Modular Multiple Dispatch with Multiple Inheritance," Eric Allen, J.J. Hallett, Victor Luchangco, Sukyoung Ryu, and Guy L. Steele Jr. SAC 2007: 22nd Annual ACM Symposium on Applied Computing

Language Features

- Components: Import other APIs but modularly checked
- Traits: Multiple inheritance of code without fields
- Objects: Leaves of type hierarchy containing fields
- Exclusive types: No object is a subtype of excluding traits.
- Functional Methods: explicit `self` parameter in the parameter list, rather than an implicit `self` parameter before the method name

```
trait Matrix excludes Vector
    opr ·(self, other: Vector): Matrix
    opr ·(other: Vector, self): Matrix
end
```

$v \cdot M + M \cdot v$

Overloading Rules

- Compare overloaded declarations pairwise.
- If any rule holds then a valid overloading:
  - Exclusion Rule
  - Subtype Rule
  - Meet Rule

Exclusion Rule

- Parameter types exclude each other.

$\texttt{trait}$ Animal $\texttt{excludes}$ Flower $\texttt{end}$

$$eat(who\colon \text{Animal}, what\colon \text{Flower})\colon \text{Boolean}$$
$$eat(who\colon \text{Flower}, what\colon \text{Animal})\colon \text{Boolean}$$

Subtype Rule

- Parameter type of one declaration is a subtype of the other.
- Return types must also be in subtype relation.

$$characteristic(x\colon \mathrm{Flower})\colon \mathrm{Object}$$
$$characteristic(x\colon \mathrm{Rose})\colon \mathrm{Thorny}$$

Meet Rule for Functions

- Exists a declaration that is more specific than both.

$$countRoses(x\colon \mathrm{Flower}, y\colon \mathrm{Rose}) = 1$$
$$countRoses(x\colon \mathrm{Rose}, y\colon \mathrm{Flower}) = 1$$
$$countRoses(x\colon \mathrm{Rose}, y\colon \mathrm{Rose}) = 2$$

Meet Rule for Functional Methods (I)

- Treating functional methods like functions is too restrictive.

trait Flower

    name(**self**)

end

trait Thorny

    name(**self**)

end

Meet Rule for Functional Methods (I)

- Treating functional methods like functions is too restrictive.

```
trait Flower
    name(self)
end

trait Thorny
    name(self)
end

object Rose extends { Flower, Thorny }
    name(self) = "Rose"
end
```

Meet Rule for Functional Methods (II)

- Ambiguity due to `self` parameter position

$\texttt{object}$ Rose $\texttt{extends}$ Flower

$\quad countRoses(\texttt{self}, l\colon \text{Lily}) = 1$

$\texttt{end}$

$\texttt{object}$ Lily $\texttt{extends}$ Flower

$\quad countRoses(r\colon \text{Rose}, \texttt{self}) = 1$

$\texttt{end}$

$countRoses(\text{Rose}, \text{Lily})$

Meet Rule for Functional Methods (II)

- Ambiguity due to `self` parameter position

$\texttt{object}$ Rose $\texttt{extends}$ Flower

$\quad$ $countRoses(\texttt{self}, l\colon \text{Lily}) = 1$

$\texttt{end}$

$\texttt{object}$ Lily $\texttt{extends}$ Flower

$\quad$ $countRoses(r\colon \text{Rose}, \texttt{self}) = 1$

$\texttt{end}$

$countRoses(\text{Rose}, \text{Lily})$

- Any trait or object declaration that provides both also provides a declaration that is more specific than both.
- `self` parameters must be in the same position.

Overloading Resolution Proof

Theorem 1. If all the overloaded declarations satisfy the static overloading rules, there are no ambiguous nor undefined calls at run time.

How about Generic Functionals?

- Overloaded declarations must have static parameters that are identical (up to α-equivalence).

$$first[\![\mathrm{T1}, \mathrm{T2}, \mathrm{T3}]\!]\big(x\colon (\mathrm{T1}, \mathrm{T2})\big)\colon \mathrm{T1} = \texttt{do}\ (a, \_) = x; a\ \texttt{end}$$
$$second[\![\mathrm{T1}, \mathrm{T2}, \mathrm{T3}]\!]\big(x\colon (\mathrm{T1}, \mathrm{T2})\big)\colon \mathrm{T2} = \texttt{do}\ (\_, b) = x; b\ \texttt{end}$$

$$first[\![\mathrm{T1}, \mathrm{T2}, \mathrm{T3}]\!]\big(x\colon (\mathrm{T1}, \mathrm{T2}, \mathrm{T3})\big)\colon \mathrm{T1} = \texttt{do}\ (a, \_, \_) = x; a\ \texttt{end}$$
$$second[\![\mathrm{T1}, \mathrm{T2}, \mathrm{T3}]\!]\big(x\colon (\mathrm{T1}, \mathrm{T2}, \mathrm{T3})\big)\colon \mathrm{T2} = \texttt{do}\ (\_, b, \_) = x; b\ \texttt{end}$$
$$third[\![\mathrm{T1}, \mathrm{T2}, \mathrm{T3}]\!]\big(x\colon (\mathrm{T1}, \mathrm{T2}, \mathrm{T3})\big)\colon \mathrm{T3} = \texttt{do}\ (\_, \_, c) = x; c\ \texttt{end}$$

More Generic Functionals

$$\texttt{trait}\ \text{Number}$$

$$\quad \ldots$$

$$\quad \texttt{opr}\ \cdot(\texttt{self}, b\colon \text{Number})\colon \mathbb{R}64$$

$$\texttt{end}$$

$$\texttt{opr}\ \cdot[\![\, T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n\,]\!]$$
$$\quad (me : \text{Vector}[\![T, n]\!], other : \text{Vector}[\![T, n]\!]) : T$$

$$\texttt{opr}\ \cdot[\![\, T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n\,]\!]$$
$$\quad (other : T, me : \text{Vector}[\![T, n]\!]) : \text{Vector}[\![T, n]\!]$$

$$\texttt{opr}\ \cdot[\![\, T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n, \texttt{nat}\ m, \texttt{nat}\ p]\!]$$
$$\quad (me\colon \text{Matrix}[\![T, n, m]\!], other\colon \text{Matrix}[\![T, m, p]\!])\colon \text{Matrix}[\![T, n, p]\!]$$

$$\ldots$$

More Features to Prove

- Generic overloaded functionals
- Where clauses
- Coercions
- Type inferene
- Self-type idiom[a]
- Pattern matching
- . . .

[a] http://projectfortress.sun.com/Projects/Community/blog/category/SelfTypes

Sukyoung Ryu

sryu@cs.kaist.ac.kr
http://plrg.kaist.ac.kr