Object-Oriented Programming in Fortress

Eric Allen
Sun Microsystems, Inc.

Joint work with:
David Chase, Christine Flood, Joseph Hallett, Victor Luchangco, Jan-Willem Maessen,
Sukyoung Ryu, Guy L. Steele Jr., Sam Tobin-Hochstadt

Project Fortress

- New language designed for high-performance computing with high programmer productivity
- Parallel features built into the core of the language

Project Status

- Draft specification and preliminary open source release available
- BSD license
- http://research.sun.com/projects/plrg

Traits

- In Fortress, traits are like "interfaces with code"
  - No fields
  - Multiple inheritance
  - Methods can contain definitions

`trait` List `end`

```
trait List
    cons(first′: ℤ64): List
    append(rest′: List): List
end
```

```
trait List
    cons(first': ZZ64): List
    append(rest': List): List
end
```

```
trait List
    cons(first′: ℤ64): List
    append(rest′: List): List
end
```

Objects

- At the leaves of the trait hierarchy are *object definitions*
- Objects define fields and methods
- *Singleton objects* define a single instance
- *Parametric objects* define constructors

```
trait List
    cons(first: ℤ64): List
    append(rest′: List): List
end

object Empty extends List
    cons(first′: ℤ64): List = Cons(first′, self)
    append(rest′: List): List = rest′
end

object Cons(first: ℤ64, rest: List) extends List
    cons(first′: ℤ64): List = Cons(first′, self)
    append(rest′: List): List = rest.append(rest′).cons(first)
end
```

```
trait List
    cons(first: ℤ64): List
    append(rest′: List): List
end

object Empty extends List
    cons(first′: ℤ64): List = Cons(first′, self)
    append(rest′: List): List = rest′
end

object Cons(hidden first: ℤ64, rest: List) extends List
    getter first(): ℤ64 = first
    cons(first′: ℤ64): List = Cons(first′, self)
    append(rest′: List): List = rest.append(rest′).cons(first)
end
```

```
trait List
    cons(first: ℤ64): List
    append(rest′: List): List
end

object Empty extends List
    cons(first′: ℤ64): List = Cons(first′, self)
    append(rest′: List): List = rest′
end

object Cons(first: ℤ64, rest: List) extends List
    cons(first′: ℤ64): List = Cons(first′, self)
    append(rest′: List): List = rest.append(rest′).cons(first)
end
```

```
trait List
    cons(first: ℤ64): List
    append(rest′: List): List
end

object Empty extends List
    cons(first′: ℤ64): List = Cons(first′, self)
    append(rest′: List): List = rest′
end

object Cons(settable first: ℤ64, rest: List) extends List
    cons(first′: ℤ64): List = Cons(first′, self)
    append(rest′: List): List = rest.append(rest′).cons(first)
end
```

```
trait List
    cons(first: ℤ64): List
    append(rest′: List): List
end

object Empty extends List
    cons(first′: ℤ64): List = Cons(first′, self)
    append(rest′: List): List = rest′
end

object Cons(first: ℤ64, rest: List) extends List
    cons(first′: ℤ64): List = Cons(first′, self)
    append(rest′: List): List = rest.append(rest′).cons(first)
end
```

Functional Methods

- Method declarations can specify that the *self* parameter occurs in an ordinary parameter position
- Effectively, such methods define new top-level functions

```
trait List
    cons(first′: ℤ64, self): List
    append(self, rest′: List): List
end

object Empty extends List
    cons(first′: ℤ64, self): List = Cons(first′, self)
    append(self, rest′: List): List = rest′
end

object Cons(first: ℤ64, rest: List) extends List
    cons(first′: ℤ64, self): List = Cons(first′, self)
    append(self, rest′: List): List = cons(first, append(rest, rest′))
end
```

```
trait List
    cons(first′: ℤ64, self): List = Cons(first′, self)
    append(self, rest′: List): List
end

object Empty extends List
    append(self, rest′: List): List = rest′
end

object Cons(first: ℤ64, rest: List) extends List
    append(self, rest′: List): List = cons(first, append(rest, rest′))
end
```

Additional Type Constraints

- *excludes clauses* ensure that two types do not share a subtype
- *comprises clauses* restrict the set of immediate subtypes of a type

```
trait List excludes {ℤ64}
    cons(first′: ℤ64, self): List = Cons(first′, self)
    append(self, rest′: List): List
end

object Empty extends List
    append(self, rest′: List): List = rest′
end

object Cons(first: ℤ64, rest: List) extends List
    append(self, rest′: List): List = cons(first, append(rest, rest′))
end
```

```
trait List excludes {ℤ64} comprises {Empty, Cons}
    cons(first′: ℤ64, self): List = Cons(first′, self)
    append(self, rest′: List): List
end

object Empty extends List
    append(self, rest′: List): List = rest′
end

object Cons(first: ℤ64, rest: List) extends List
    append(self, rest′: List): List = cons(first, append(rest, rest′))
end
```

Value Traits and Value Objects

- A *value* object must not contain mutable fields
- A value trait must be extended only by other value traits and value objects
- Value objects can be copied
- Copies are identified

```
value trait List excludes {ℤ64} comprises {Empty, Cons}
    cons(first′: ℤ64, self): List = Cons(first′, self)
    append(self, rest′: List): List
end

value object Empty extends List
  append(self, rest′: List): List = rest′
end

value object Cons(first: ℤ64, rest: List) extends List
  append(self, rest′: List): List = cons(first, append(rest, rest′))
end
```

Generic Types

- Trait and object definitions (and method and function defintions) can be parametric
- Parametric instantiations are retained at runtime

$$\texttt{value trait } \mathrm{List}[\![T]\!] \texttt{ excludes } \{T\} \texttt{ comprises } \{\mathrm{Empty}[\![T]\!], \mathrm{Cons}[\![T]\!]\}$$
$$\quad \mathit{cons}(\mathit{first}' \colon T, \texttt{self}) \colon \mathrm{List}[\![T]\!] = \mathrm{Cons}(\mathit{first}', \texttt{self})$$
$$\quad \mathit{append}(\texttt{self}, \mathit{rest}' \colon \mathrm{List}[\![T]\!]) \colon \mathrm{List}[\![T]\!]$$
$$\texttt{end}$$

$$\texttt{value object } \mathrm{Empty}[\![T]\!] \texttt{ extends } \mathrm{List}[\![T]\!]$$
$$\quad \mathit{append}(\texttt{self}, \mathit{rest}' \colon \mathrm{List}[\![T]\!]) \colon \mathrm{List}[\![T]\!] = \mathit{rest}'$$
$$\texttt{end}$$

$$\texttt{value object } \mathrm{Cons}[\![T]\!](\mathit{first} \colon T, \mathit{rest} \colon \mathrm{List}[\![T]\!]) \texttt{ extends } \mathrm{List}[\![T]\!]$$
$$\quad \mathit{append}(\texttt{self}, \mathit{rest}' \colon \mathrm{List}[\![T]\!]) \colon \mathrm{List}[\![T]\!] = \mathit{cons}(\mathit{first}, \mathit{append}(\mathit{rest}, \mathit{rest}'))$$
$$\texttt{end}$$

```
value trait List⟦T⟧ excludes {T} comprises {Empty⟦T⟧, Cons⟦T⟧}
    cons(first′: T, self): List⟦T⟧ = Cons(first′, self)
    append(self, rest′: List⟦T⟧): List⟦T⟧
end

value object Empty⟦T⟧ extends List⟦T⟧
   append(self, rest′: List⟦T⟧): List⟦T⟧ = rest′
end

value object Cons⟦T⟧(first: T, rest: List⟦T⟧) extends List⟦T⟧
   append(self, rest′: List⟦T⟧): List⟦T⟧ = cons(first, append(rest, rest′))
end

object ListBox⟦T⟧(settable elts : List⟦T⟧) end
```

Wrapped Fields

- *Forwarding methods* are implicitly declared for all the methods in the static type of the wrapped field
- Implicit forwarding methods are shadowed by explicit definitions

```
value trait List⟦T⟧ excludes {T} comprises {Empty⟦T⟧, Cons⟦T⟧}
    cons(first′: T, self): List⟦T⟧ = Cons(first′, self)
    append(self, rest′: List⟦T⟧): List⟦T⟧
end

value object Empty⟦T⟧ extends List⟦T⟧
   append(self, rest′: List⟦T⟧): List⟦T⟧ = rest′
end

value object Cons⟦T⟧(first: T, rest: List⟦T⟧) extends List⟦T⟧
   append(self, rest′: List⟦T⟧): List⟦T⟧ = cons(first, append(rest, rest′))
end

object ListBox⟦T⟧(settable wrapped elts : List⟦T⟧) end
```

Coercions

- A type S can define a coercion *from* another type T
- This allows instances of T to be coerced and applied in contexts requiring an S

```
value trait List⟦T⟧ excludes {T} comprises {Empty⟦T⟧, Cons⟦T⟧}
    cons(first′: T, self): List⟦T⟧ = Cons(first′, self)
    append(self, rest′: List⟦T⟧): List⟦T⟧
end

value object Empty⟦T⟧ extends List⟦T⟧
   append(self, rest′: List⟦T⟧): List⟦T⟧ = rest′
end

value object Cons⟦T⟧(first: T, rest: List⟦T⟧) extends List⟦T⟧
   append(self, rest′: List⟦T⟧): List⟦T⟧ = cons(first, append(rest, rest′))
end

object ListBox⟦T⟧(settable elts : List⟦T⟧)
   coercion (xs: List⟦T⟧) = ListBox⟦T⟧(xs)
end
```

Hidden Type Variables

- Type definitions can be universally quantified with respect to additional, hidden, type variables that *are not type parameters*

$$\texttt{value trait } \mathrm{List}[\![T]\!] \texttt{ excludes } \{T\} \texttt{ comprises } \{\mathrm{Empty}, \mathrm{Cons}[\![T]\!]\}$$
$$\quad \mathit{cons}(\mathit{first}' \colon T, \texttt{self}) \colon \mathrm{List}[\![T]\!] = \mathrm{Cons}(\mathit{first}', \texttt{self})$$
$$\quad \mathit{append}(\texttt{self}, \mathit{rest}' \colon \mathrm{List}[\![T]\!]) \colon \mathrm{List}[\![T]\!]$$
$$\texttt{end}$$

$$\texttt{value object } \mathrm{Empty} \texttt{ extends } \mathrm{List}[\![T]\!] \texttt{ where } \{T \texttt{ extends } \mathrm{Object}\}$$
$$\quad \mathit{append}(\texttt{self}, \mathit{rest}' \colon \mathrm{List}[\![T]\!]) \colon \mathrm{List}[\![T]\!] = \mathit{rest}'$$
$$\texttt{end}$$

$$\texttt{value object } \mathrm{Cons}[\![T]\!](\mathit{first} \colon T, \mathit{rest} \colon \mathrm{List}[\![T]\!]) \texttt{ extends } \mathrm{List}[\![T]\!]$$
$$\quad \mathit{append}(\texttt{self}, \mathit{rest}' \colon \mathrm{List}[\![T]\!]) \colon \mathrm{List}[\![T]\!] = \mathit{cons}(\mathit{first}, \mathit{append}(\mathit{rest}, \mathit{rest}'))$$
$$\texttt{end}$$

```
value trait List⟦T extends U⟧ extends List⟦U⟧ where {U extends Object}
    excludes {T}
    comprises {Empty, Cons⟦T⟧}
    cons(first′: U, self): List⟦U⟧ = Cons(first′, self)
    append(self, rest′: List⟦U⟧): List⟦U⟧
end

value object Empty extends List⟦T⟧ where {T extends Object}
   append(self, rest′: List⟦T⟧): List⟦T⟧ = rest′
end

value object Cons⟦T extends U⟧(first: T, rest: List⟦T⟧) extends List⟦U⟧
   where {U extends Object}
   append(self, rest′: List⟦U⟧): List⟦U⟧ = cons(first, append(rest, rest′))
end
```

Top-Level Functions

- Functions defined at top-level allow additional functionality to be added to existing types
- Overloaded functions are resolved via multiple dynamic dispatch
- Lack of runtime ambiguity is statically ensured

$$length[\![T]\!](xs\colon \mathrm{List}[\![T]\!])\colon \mathbb{Z}32$$
$$length[\![T]\!](xs\colon \mathrm{Empty}) = 0$$
$$length[\![T]\!](xs\colon \mathrm{Cons}[\![T]\!]) = 1 + length(xs.rest)$$

$$
\begin{array}{l}
nth[\![T]\!](n\colon \mathbb{Z}32, xs\colon \mathrm{List}[\![T]\!])\colon T \\
\quad \texttt{requires}\ \{\, 0 \le n < length(xs)\,\} \\
nth[\![T]\!](n\colon \mathbb{Z}32, xs\colon \mathrm{Empty}[\![T]\!]) = \texttt{throw}\ \ new\ \mathrm{UncheckedException}() \\
nth[\![T]\!](n\colon \mathbb{Z}32, xs\colon \mathrm{Cons}[\![T]\!]) = \\
\quad \texttt{if}\ \ n = 0\ \texttt{then}\ \ xs.first \\
\quad \texttt{else}\ \ nth(n-1, xs.rest)\ \texttt{end}
\end{array}
$$

```
nth⟦T⟧(n: ℤ32, xs: List⟦T⟧): T
  requires { 0 ≤ n < length(xs) } = do
  typecase xs of
    Empty⟦T⟧ ⇒ throw new UncheckedException()
    else⇒ if n= 0 then xs.first
                  else nth(n − 1, xs.rest) end
  end
end
```

Components and APIs

- Independent program pieces are wrapped in *component definitions*
- Components import and export *APIs*
- Matching components can be linked
- Compound components can be upgraded with new matching constituents
- Upgrades of one component do not affect another

```
component Lists
export Lists

value trait List⟦T extends U⟧ extends List⟦U⟧ where {U extends Object}
    excludes {T}
    comprises {Empty, Cons⟦T⟧}
    cons(first′: U, self): List⟦U⟧ = Cons(first′, self)
    append(self, rest′: List⟦U⟧): List⟦U⟧
end

value object Empty extends List⟦T⟧ where {T extends Object}
   append(self, rest′: List⟦T⟧): List⟦T⟧ = rest′
end

value object Cons⟦T extends U⟧(first: T, rest: List⟦T⟧) extends List⟦U⟧
   where {U extends Object}
   append(self, rest′: List⟦U⟧): List⟦U⟧ = cons(first, append(rest, rest′))
end

end
```

```
api Lists

value trait List⟦T extends U⟧ extends List⟦U⟧ where {U extends Object}
    excludes {T}
    comprises {Empty, Cons⟦T⟧}
    cons(first′: U, self): List⟦U⟧
    append(self, rest′: List⟦U⟧): List⟦U⟧
end

value object Empty extends List⟦T⟧ where {T extends Object} end

value object Cons⟦T extends U⟧(first: T, rest: List⟦T⟧) extends List⟦U⟧
  where {U extends Object}
end

end
```

```
component ListBoxes
import Lists
export ListBoxes

object ListBox⟦T⟧(settable elts : List⟦T⟧)
   coercion (xs: List⟦T⟧) = ListBox⟦T⟧(xs)
end

end
```

```
api ListBoxes
import Lists

object ListBox[[T]](settable elts : List[[T]])
   coercion (xs: List[[T]])
end

end
```

compile Lists.*fss*
compile ListBoxes.*fss*

link MyProgram `from` ListBoxes `with` Lists

compile Lists.*fss*
compile ListBoxes.*fss*

link MyProgram `from` ListBoxes `with` Lists

upgrade MyProgram `from` MyProgram `with` NewListBoxes

Summary

- Trait-based object system
- Generic types with runtime existence and hidden type variables
- Top-level functions with multiple dispatch
- Encapsulated upgradable components

http://research.sun.com/projects/plrg