Project Fortress Reference Card

Installation

Prerequisites

Java 1.6 (or later) and Ant 1.6 (or later)

Fortress distribution

By a zip distribution:

`http://projectfortress.sun.com/Projects/Community/downloads`

or an svn distribution:

```
svn co https://projectfortress.sun.com/svn/Community/trunk PFC
```

Unzip the zip file or follow the instructions in `README.txt`.

Hello, World!

```
component hello
export Executable
run() = println("Hello, World!")
end
```

A component must have the same name as the enclosing file. To be executable, a component must export the Executable API and implement a *run* function, which takes no arguments. To run the program, put the script `PFC/bin/fortress` on your path and type the following on the command line:

```
fortress PFC/ProjectFortress/hello.fss
```

'Fortify' Code Formatter

Fortify is an Emacs-based tool which formats a Fortress program typed on ASCII keyboards for processing by LaTeX. For example, the left-hand side below is produced by the Fortify tool and LaTeX from the right-hand side:

```
sum:ℝ64 := 0          sum: RR64 := 0
for k ← 1:n do        for k<-1:n do
  a_k := (1-α)b_k       a[k] := (1-alpha)b[k]
  sum += c_k x^k        sum += c[k] x^k
end                   end
```

In addition, the following symbols are produced by the ASCII character sequences below them:

⟦	⟧	∑	∏	⟨	⟩	↦	∈	∪	t_2
`[\`	`\]`	`SUM`	`PROD`	`<|`	`|>`	`|->`	`IN`	`UNION`	`t2`

Fortress Language Syntax

Concrete examples of various pieces of the language syntax are presented in this document. For the entire syntax including expressions and declarations of trait, object, variable, function, field, and method, please refer to the Fortress language specification:

`research.sun.com/projects/plrg/fortress.pdf`

Function Declaration

A function may take no parameters or multiple parameters. The return type annotation after the parameter may be omitted.

```
fibonacci(n: ZZ32): ZZ32 =
  if n < 0
  then fail "Non-negative integer is expected."
  elif n < 2 then n
  else fibonacci(n-1) + fibonacci(n-2)
  end
run() = println (fibonacci 5)
```

Juxtaposition

Juxtaposition may be a function call, a multiplication, or a string concatenation depending on the types of the juxtaposed items as defined in the libraries. When in doubt, parenthesize.

$$u = n(n+1)\sin 3n\, x \log\log y$$
$$w = (n(n+1))(\sin(3n)x)(\log(\log y))$$

Operator Declaration

An operator may be postfix, prefix, infix, nofix, multifix, or enclosing. The fixity of the operator is determined by the context: `5!`, `-3`, `8/7`, `:`, or `|-9|`. Certain infix operators may be chained: `0 <= n <= 1`.

```
opr (n: ZZ32)! =
  if 0 <= n <= 1 then 1
  elif n > 1 then n (n-1)!
  else fail "Non-negative integer is expected."
  end
run() = assert(5!, 120)
```

Conservative Operator Precedence

- If it is traditional in math, it works:
  $a+b>c \quad b\le a\wedge a<d \quad a\cup b\cap c=d$
- Operator precedence is not transitive: NO $a+b\vee c$
- Spacing must match precedence: NO $a+b\ /\ c+d$
- Different areas of math don't mix: NO $a+b\cup c$

Potentially Parallel Constructs

- Tuples: `(a,b,c) = (f(x),g(y),h(z))`
- Parallel blocks: `do foo(a) also do foo(b) end`
- Functions, operators, method call recipients, and their arguments: `fail "Division by zero"`, `13-5`, `receiver.method(v)`
- Expressions with generators:

```
> SUM[x <- xs, y <- ys] x y
```
$$\sum_{\substack{x\leftarrow xs\\ y\leftarrow ys}} x\,y$$

```
> <| x^2 | x <- xs, x > 43 |>
```
$\langle x^2 \mid x \leftarrow xs, x > 43 \rangle$

```
> prime[i] := false, i <- p^2:upper:p
```
$prime_i := false, i \leftarrow p^2 : upper : p$

```
> for i <- p^2:upper:p do
    prime[i] := false
  end
  for i ← p^2 : upper : p do
    prime_i := false
  end
```

```
> for l <- seq(f.lines) do println l end
  for l ← seq(f.lines) do println l end
```

Atomic Updates to Shared Locations

Every iteration of a `for` loop may be run in parallel, which may result in concurrent updates to shared locations. The `atomic` expressions perform such updates atomically. A mutable variable is declared with either the `var` modifier or `:=` instead of `=`, and it must be declared with its type.

```
opr (n: ZZ32)! = do
  var result: ZZ32 = 1
  for i <- 1:n do atomic result := result i end
  result
end
```

Dynamic Overloading

- Overloading is chosen based on the run-time types of the arguments.
- A pair of overloadings is legal if:
  - one is more specific than the other,
  - they are provably disjoint, or
  - another overloading is more specific than both of them
- Ambiguity is detected at compile time. Because the first and the second declarations below are incomparable, assuming Number is a supertype of $\mathbb{Z}32$, the third declaration is required to resolve the ambiguity:

  $foo(a\colon \text{Number}, b\colon \mathbb{Z}32)\colon \mathbb{Z}32$
  $foo(a\colon \mathbb{Z}32, b\colon \text{Number})\colon \mathbb{Z}32$
  $foo(a\colon \mathbb{Z}32, b\colon \mathbb{Z}32)\colon \mathbb{Z}32$

Traits and Objects

- Traits are like interfaces in Java™ code. Traits may have `comprises` and `excludes` clauses and they do not have fields.
- Objects are like final classes in Java™ code. They may contain field declarations and may be parameterized, but they cannot be extended.
- Traits and objects may be generic and may extend multiple traits.

```
trait Exception
    comprises { UncheckedException,
                CheckedException }
end
trait UncheckedException
    extends Exception
    excludes CheckedException
end
object FailCalled(s: String)
    extends UncheckedException
  getter asString(): String = "FAIL: " s
end

fail[\T\](s:String):T = throw FailCalled(s)
```

Comments

```
(* comment *)
(*) end-of-line comment
```

File Input & Output

```
import File.{...}
import Writer.{...}
fin: FileReadStream := FileReadStream input
fout: WriteStream = Writer output
names(): Generator[\String\] = fin.lines()
fout.println "Done!"
fin.close()
fout.close()
```

Ranges

`2:5`	`3#4`	`1:10:2`
$\{2,3,4,5\}$	$\{3,4,5,6\}$	$\{1,3,5,7,9\}$

Arrays

Arrays of fixed size at *compile time*

```
p: ZZ32[1000] = array1[\ZZ32,1000\](0)
```

Arrays of varying size at *run time*

```
p: Array[\ZZ32,ZZ32\] = array[\ZZ32\](size).fill(0)
```

Array indexing

`a[i] b[i,j] c[i,j,k]`	array indexing
`a[0:10] b[#(20,10)]`	subarray extraction

Sets

```
import Set.{...}
p = {[\ZZ32\] 2, 3, 5, 7, 11, 13, 17, 19}
s = {[\ZZ32\] d | d <- 1:n, d|n }
(t, u, v) = (p UNION s, p INTERSECTION s,
             p DIFFERENCE s)
BIG UNION[i <- p] { (i, i.asString) }
```

Lists

Like arrays, 1-dimensional indexing works on lists.

```
import List.{...}
fibs = <|[\ZZ32\] 1, 1, 2, 3, 5, 8, 13 |>
<|[\ZZ32\] d | d <- 1:n, d|n |>
fibs[0#5] = <|[\ZZ32\] 1, 1, 2, 3, 5 |>
println(empty || strings)
noHello(xs:List[\String\]):List[\String\] =
  xs.filter(fn (x) => x =/= "Hello")
```

Maps

Keys in a map aggregate constant or a map comprehension must be distinct. `UPLUS` performs a disjoint union and throws the `KeyOverlap` exception when a key occurs in both maps.

```
import Map.{...}
{1 |-> "one", 2 |-> "two", 3 |-> "three"}
{p |-> p^2 | p <- primes}
BIG UNION[(v,k) <- pairs]
  { k asif String |-> v asif ZZ32 }
BIG UPLUS[(v,k) <- pairs]
  { k asif String |-> v asif ZZ32 }
```

Command Line Arguments

`trips.fss`

```
component trips
import System.args
export Executable
run() = if |args| = 0
        then println "Number is expected."
        else println args[0] end
end
```

Command line

```
fortress trips.fss 50
```

Command Line Tools

Several command line tools are available for the Fortress developers. For example, one can generate an API from a component as follows:

```
fortress api [-out file] somefile.fss
```

Typing `fortress` prints the usage information.

Resources

The Project Fortress community website includes more information including learning material, the language specification, several mailing lists, and examples: `http://projectfortress.sun.com`