

Fortress Macros

Jon Rafkind
Ryan Culpepper

Programming Language Research Group

The Fortress Language

A high productivity language

- Efficient use of processors
- Efficient use of programmers

Enabled by

- Lots of parallelism
- Mathematical syntax, useful types and operations, etc

Put the language in the libraries

Designers cannot anticipate every domain, every pattern, every application of Fortress.

Example: parallel processing of matrices split across multiple processors

Too complicated to support all patterns of parallelism natively, “in-language”

Language must be “growable.” Make a small but powerful core, then push complexity into libraries.

Put the syntax in the libraries

Designers cannot anticipate every notation, every domain specific language.

Example: embedded XML and SQL

Too complicated to support all notations natively, “in-language”

Provide ability to extend language’s syntax.

Quick intro to Fortress

Component based system.

Two types of files

- Api - declares function signatures and types
- Component - implements functions in an api

```
api Foobar  
end
```

```
component Blah  
exports Foobar  
end
```

Hello world

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

Composability

Macros are defined in apis and should compose well together

```
api Xml
```

```
  grammar XmlLang
```

```
    Xml Expr :=
```

```
      s : XmlStuff  $\Rightarrow$  < [ XmlNode(s) ] >
```

```
api Yaml
```

```
  import Xml.XmlLang
```

```
  grammar YamlLang extends Xml
```

```
    yaml Expr :=
```

```
      id : Id : value : Expr  $\Rightarrow$  < [ < id > value < id > ] >
```

Recursive macros

How to handle recursive macros?

```
or x xs ... = if x then x else or xs end
```

How can we define the parser including **or** if we need the parser to parse the definition of **or**?

Insight: We don't need the template to parse occurrences of **or**, just to generate code for them.

Parsing vs Transformation

Perform macro expansion in two steps

- Parse
- Transform

The parser can be created without any knowledge of transformation

Rats!

Accepts parsing expression grammars (PEG)

Produces packrat parsers (Ford 2004)

Example:

```
String foobar :=
```

```
hello bob { yyValue = "hello bob"; }
```

```
/ hello gary { yyValue = "hello gary"; };
```

Rats! modules

Productions packed into a module

```
module com.sun.fortress.parser.Expression;  
body{  
Expr Expr =  
    seed:ExprFront list:ExprTail*  
    { yyValue = (Expr)apply(list, seed); };  
}
```

Parsing

somemacro *v* : Expr \Rightarrow < [*println* "Hello " *v*] >

=>

```
(SyntaxDef
  (SyntaxSymbols
    (item "somemacro")
    (PrefixedSymbol (item "v")))
  (PreTransformer "println ..."))
```

Parsing

Parser creates nodes with placeholder "transformer names" for templates

or $xs^* \Rightarrow$

case xs of

Empty $\Rightarrow \langle [false] \rangle$

Cons(h, t) $\Rightarrow \langle [if\ h\ then\ h\ else\ or\ t^*\ end] \rangle$

end

Action routine in the parser becomes

or $xs:Expr^*\{$

return new Transformer("orT123", xs);

}

Parsing

Create a new Rats! module with the user defined productions

```
module USER;  
body{  
Expr Or =  
  xs:Expr* {  
    yyValue = new Transformer("orT123", xs);  
  }  
}
```

Also add Or to the original Expr production

```
Expr Expr =  
  Or / seed:ExprFront list:ExprTail* ...
```

Parsing

$\text{Cons}(h, t) \Rightarrow \langle [\text{if } h \text{ then } h \text{ else or } t^* \text{ end}] \rangle$

Use new parser to parse the template

orT =

```
(if (condition h)
    (then h)
    (else (Transformer "orT123" t)))
```

```
transformers["orT123"] = orT;
```

Transformation

- Use ideas from Scheme: (ellipses, hygiene, ...)
- Case expressions match input to a constructor and invoke the corresponding transformer
- Replace transformers with their bodies, substituting pattern variables along the way

Transformation Mechanism

- Transformer bodies contain applications, case expressions, and variable references
- Interpreter for first-order functional language

```

transform(syntax) =
  typecase syntax of
    Node ⇒ syntax
    PatternVariable ⇒ environment(syntax .name)
    Transformer ⇒ do
      environment = syntax .vars
      transform(lookupBody(syntax .name))
    end
  end
end

```

Extensible parsers

Rats! parsers can be extended by adding new choices to a production

Example:

```
Node Expr := ExprFront / OtherStuff
```

Add users syntax

```
Node Expr := USER_Expr / ExprFront / OtherStuff
```

If the users syntax doesn't match, the original Expr will be used

Ellipses

or $xs : \text{Expr} * \dots$

xs is not `Expr` but rather `EllipsesExpr`

`(... (EllipsesExpr ...) ...)`

Use Macro by Example algorithm Kohlbecker and Wand[87]

For

Expr | Expr :=

a:for {i:Id <- e:Expr ,? SPACE}* ; b:do block:Expr
 ; c:end =>

<[for2 i** ; e** ; do block ; end]>

| a:for2 i:Id* ; e:Expr* ; b:do block:Expr ; c:end =>

case i of

Empty => <[block]>

Cons(ia, ib) =>

case e of

Cons (ea, eb) =>

<[((ea).loop(fn ia =>

(for2 ib** ; eb** ; do block ; end)))]>

For

```
for  $a \leftarrow n, q \leftarrow b$ ; do  
  (println  $a$  “ and ”  $q$ );  
end
```

Becomes

```
 $(n.loop(\mathbf{fn} (a) \Rightarrow (b.loop(\mathbf{fn} (q) \Rightarrow (\mathit{println} a \text{ “ and ” } q))))))$ 
```

Thank you

Questions?