

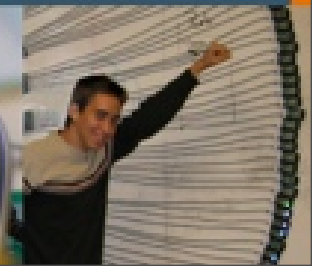
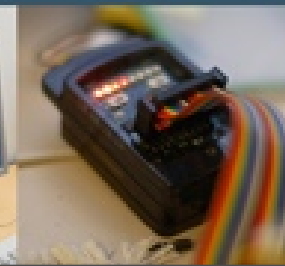


The Fortress Static End

Eric Allen



2008
Sun Labs
Open House



Copyright 2008 Sun Microsystems, Inc. ("Sun").

All rights are reserved by Sun except as expressly stated as follows. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers, or to redistribute to lists, requires prior specific written permission of Sun.

Visitors

```
abstract class List {  
    abstract <T> T accept(ListVisitor<T> that);  
}
```

```
class Empty extends List {  
    <T> T accept(ListVisitor<T> that) {  
        return that.forEmpty(this);  
    }  
}
```

Visitors

```
class Cons extends List {
    public Object car;
    public List cdr;
    Cons(Object _car, List _cdr) {
        car = _car;
        cdr = _cdr;
    }
    <T> T accept(ListVisitor<T> that) {
        return that.forCons(this);
    }
}
```

Visitors

```
interface ListVisitor<T> {  
    public T forEmpty(Empty that);  
    public T forCons(Cons that);  
}
```

Depth-First Visitors

- A depth first visitor is a concrete class implementing the Visitor interface
- Each for- method recursively calls the visitor on all its fields, and calls a corresponding *forOnly- method* with the results
- You can subclass and override the for- and forOnly- methods of the default visitor to concisely write very powerful visitors

```
abstract class ListDepthFirstVisitor<T> extends ListVisitor<T> {  
    public T forEmpty(Empty that) {  
        return forEmptyOnly(that);  
    }  
    public T forCons(Cons that) {  
        return forConsOnly(that, that.first,  
                             that.rest.accept(this));  
    }  
    public abstract forDefaultCase(List that);  
  
    public T forEmptyOnly(Empty that) {  
        return forDefaultCase(that);  
    }  
    public T forConsOnly(Cons that,  
                        Object first_result,  
                        List rest_result)  
    { return forDefaultCase(that); }  
}
```

Now I can write my own depth-first visitor:

```
class CopyVisitor extends ListDepthFirstVisitor<List> {  
    public List forDefaultCase(List that) { return that; }
```

```
    List forConsOnly(Cons that,  
                    Object first_result,  
                    List rest_result)  
    {  
        return new Cons(first_result, rest_result);  
    }  
}
```

Advantages

- Recursive results are readily available as parameters (similar to pattern matching)
- If most of the for- methods for your visitor simply recur on the subcomponents and assemble the results, you can save a lot of space
- If many for- methods are irrelevant to your visitor, the defaultCase method can save a lot of space

Tips

- When you need to first do some computation before recurring in a particular case, just override the for- method
- For Fortress depth-first visitors, it's helpful to copy all the code from NodeDepthFirstVisitor into your class, commented
- This will help you quickly retrieve the right signatures

Other Helpful Autogenerated Visitors

- **NodeAbstractVisitor:** An abstract class with a default case. Each for- method delegates to the for- method of the corresponding superclass. The for- method of AbstractNode delegates to defaultCase
- Subclass and override a for- method for a particular type to affect handling of that type and all its subtypes

Other Helpful Autogenerated Visitors

- NodeUpdateVisitor: Subclass of NodeDepthFirstVisitor
- Each forOnly method checks the recursive results passed to it
- If any of them have changed, it generates a new node containing the recursive results
- Otherwise, it returns the original result

com.sun.fortress.compiler.Fortress.analyze

- Given a GlobalEnvironment of APIs, a set of APIs to compile, and a set of components to compile
- First handles APIs
- Then handles components

First Handle APIs

- > Build ApilIndices (wrappers over API Nodes)
- > Build a new GlobalEnvironment that includes all the APIs we are compiling
- > Disambiguate our given APIs so they contain only fully qualified names
- > Rebuild ApilIndices
- > Rebuild our GlobalEnvironment
- > ...

First Handle APIs

- Rewrite Grammars
- Rebuild ApiIndices
- Rebuild GlobalEnvironment
- Do static checks on APIs (currently, very little)
- Write out code to a “repository” object

Rewriting Grammars

- New grammar definitions can be written in APIs
- Various processing steps must be done on these definitions
- Ultimately, they are compiled into a parser that extends the core Fortress parser
- Janus Nielsen (Denmark) has been driving syntax abstraction

Then Handle Components

- Build ComponentIndices
- Disambiguate components so that all type references and most variable references are fully qualified
- Rebuild ComponentIndices
- Statically check all components
- *Optimization phases can be inserted here*

Disambiguation Steps

- Disambiguation is handled primarily by two classes:
 - > `com.sun.fortress.compiler.disambiguator.TypeDisambiguator`
 - > `com.sun.fortress.compiler.disambiguator.ExprDisambiguator`
- Both extend `NodeUpdateVisitor`

Static Checking is Driven by:

`com.sun.fortress.compiler.StaticChecker`

- Right now, all this does is iterate over all components and call the TypeChecker visitor on them
- Right now, the TypeChecker visitor is only called when testing those tests in the `static_tests` directory

The TypeChecker Visitor

`com.sun.fortress.compiler.typechecker.TypeChecker`

- This is a subclass of `NodeDepthFirstVisitor`
- It contains the following fields:
 - > `private TraitTable table;`
 - > `private StaticParamEnv staticParamEnv;`
 - > `private TypeEnv typeEnv;`
 - > `private final CompilationUnitIndex compilationUnit;`
- `private final SubtypeChecker subtypeChecker;`
- `private final Map<Id, Option<Set<Type>>> labelExitTypes; // Note: this is mutable state.`

The TypeChecker Visitor

- When visiting a new binding node, it is necessary to extend the typeEnv
- This is done by overriding the for- method for that node with a method that constructs a new TypeChecker and uses it to call the subnodes

The TypeChecker Visitor

- When visiting a non-binding node, we need only override the `for...Only` method, which provides all the results of recurring on children as parameters

The TypeChecker Visitor

- Returns a TypeCheckerResult object, which contains:
 - > a set of static errors
 - > a type (if applicable)
 - > a rewritten AST

Type Environments

- Type Environments are immutable environments that provide various lookup and extension methods
- The type environment classes form a composite class hierarchy rooted at abstract class `TypeEnv`
- There is one concrete subclass for each type of binding node
- During a lookup, information is retrieved lazily from the binding node

Type Environments

- Look at VarTypeEnv, for example
- The only overridden method is “binding”
- All other lookup and extension functionality is defined in the superclass, TypeEnv

The SubtypeChecker

- The subtype checker takes two types and returns a set of constraints that must hold if the first type is a subtype of the other
- Right now, the only constraints it gives back are the trivial constraints **TRUE** and **FALSE**

The TypeAnalyzer

- Long term, the idea is to replace the SubtypeChecker with class TypeAnalyzer
- TypeAnalyzer generates more interesting constraints and handles more of the Fortress type system
- Right now, it is too slow, but we now know a way to make it *much* faster
- Dan Smith (Houston) is working on this

One more thing...

- Lots of the static front end makes use of the PLT Utilities libraries (an open source project from Rice University)
- This library provides lots of classes for function objects, set objects, etc., useful for functional programming
- Javadoc is available at:
 - > <http://drjava.org/javadoc/plt/>

Rendering Tools

- There are various tools for rendering Fortress code as LaTeX
- These tools generate code that you can import into a LaTeX file that also imports the Fortress macros:
- `\input{$FORTRESS_HOME/Fortify/fortify-macros.tex}`

Write a file such as the following:

```
\documentclass{article}
```

```
\input{$FORTRESS_HOME/Fortify/fortify-macros.tex}
```

```
\begin{document}
```

```
\end{document}
```

fortify

- You can format an entire Fortress file at the command line by invoking the fortify script:
 - > fortify YOUR-FILE.fss

fortify

- The result is that a formatted file is created, with the same name and at the same location as the input file, but with extension `.tex`
- **WARNING:** Make sure your original file does not have suffix `.tex` or this command will overwrite it!

Calling fortify

- Move one of your .fss files to the same directory as your LaTeX file
- Call fortify on your .fss file
- This will produce a .tex file

Adding your formatted Fortress

```
\documentclass{article}
```

```
\input{$FORTRESS_HOME/Fortify/fortify-macros.tex}
```

```
\begin{document}
```

```
\input{YOUR_FILE.tex}
```

```
\end{document}
```

Rendering the LaTeX

- Call LaTeX on your wrapper file to produce a rendered version of your code

fortex

- You can also process a Fortress file containing "doc comments"
- A Fortress doc comment consists of a series of lines, where
- The firstline begins with "(**", the last line consists of "**)" and lines in
- between begin with "*", with optional leading spaces and tabs at the beginning of each line

fortex

```
(**  
 * Takes a non-negative integer n and returns n!  
**)  
factorial(n) =  
  if n = 0 then 1  
  else n factorial(n-1) end
```

fortex

- Add some doc comments to one of your files
- Run fortex on it
- Embed the result in your wrapper file

fortex

- **WARNING:** Make sure your original file does not have suffix `.tex` or this command will overwrite it!

foreg

- It is often useful to fortify an example that is not itself a complete program
- In such cases, it's still helpful if an enclosing program can be provided to check that the example is valid
- For this purpose, the function 'foreg' is provided
- This command deletes all lines in a file except those lines surrounded by "example" delimiters

foreg

```
(* EXAMPLE *)  
your example here  
(* END EXAMPLE *)
```

foreg

- Look at:

PFC/SpecData/examples/basic/Expr.Map.fss

foreg

- Copy this file to your directory with your wrapper file
- Call foreg on it
- Import the result into your wrapper file

foreg

- **WARNING:** Make sure your original file does not have suffix `.tex` or this command will overwrite it!

fortick

- fortick takes a file name and fortifies all occurrences of text in that file delimited by ticks
- It writes the result to a new file in the same location, with suffix ".tex"
- This is useful for embedding Fortress code in a LaTeX file

fortick

- If the opening tick and closing tick are adjacent, then fortick simply replaces the pair with a single tick
- This provides an escape for those who need to use ticks in LaTeX document

fortick

- Copy your wrapper file to some other file *with a distinct name* and with suffix `.tick`
- In your new file, add some Fortress code enclosed in back ticks
- Call `fortick` on your new file; this will produce a `.tex` file
- Run LaTeX on your new `.tex` file

fortick

- **WARNING:** Make sure your original file does not have suffix `.tex` or this command will overwrite it!