Enabling Go Program Analysis in Rascal

Luke Swearngan and Mark Hills

23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2023), Engineering Track
October 2-3, 2023
Bogotá, Colombia

https://www.rascal-mpl.org/

Background: Why look at Go?

- Go is a widely used language with an interesting channel-based concurrency model plus traditional concurrency features

- Origin of this work was a student MS thesis

- Earlier work had studied how channel-based concurrency was used in Go programs (see “An Empirical Study of Message Passing Concurrency in Go Projects” by Dilley and Lange from SANER 2019)

- Student’s Focus: How do people use traditional concurrency features, like mutex and condition variables? Do they?

Image from: https://medium.com/nerd-for-tech/learn-golang-in-one-blog-fdd568e6f631

First Idea: Just write this in Go!

- Go includes several libraries for working with Go programs, so it's fairly easy to get started
  - The `go/ast` library defines all the interfaces (e.g., `Expr`) and structures (e.g., `SelectorExpr`) for Abstract Syntax Tree nodes
  - The `go/parser` library lets you parse Go code and get back an AST
  - The `go/token` library defines all the lexical tokens in the language
- So, just create a Visitor, walk the AST, and collect the info — done!

Image from: https://medium.com/nerd-for-tech/learn-golang-in-one-blog-fdd568e6f631

The problem: Matching AST nodes

NOTE: We are looking for something like: var wg sync.WaitGroup

```
func matchWaitGroupDecl(x *ast.GenDecl, v *Visitor, n ast.Node) {
    for i := 0; i < len(x.Specs); i++ {
        if spec, ok := x.Specs[i].(*ast.ValueSpec); ok == true {
            if spec.Type != nil {
                if t, ok := spec.Type.(*ast.SelectorExpr); ok == true {
                    if tsel, ok := t.X.(*ast.Ident); ok == true {
                        if tsel.Name == "sync" && t.Sel.Name == "WaitGroup" {
                            for j := 0; j < len(spec.Names); j++ {
                                id := spec.Names[j]
                                v.addDef(createDecl(id.Name, WaitGroup))
                                v.state.addWaitGroupDecl()
                            }
                        }
                    }
                }
            }
} } } } // all on one line so this fits on a slide!
```

Image from: https://medium.com/nerd-for-tech/learn-golang-in-one-blog-fdd568e6f631

The problem: Matching AST nodes

- Note: the code on the prior slide is not **bad**, it is just very verbose!

  - `spec, ok := x.Specs[I].(*ast.ValueSpec)` is a *type assertion*: we want to make sure that spec (which is just defined as being of interface type `Spec`) is of a certain concrete type (a `ValueSpec`) — this is essentially a downcast

  - We then check to see if `ok == true`, which means that the type assertion passed and `spec` can now be treated as a value of that type (which it must be if this worked) — if we just do the assertion without the `ok` check, this will panic (i.e., crash) if the assertion fails

Image from: https://medium.com/nerd-for-tech/learn-golang-in-one-blog-fdd568e6f631

Is there a better way?

- Rascal is designed for these kinds of applications!

- The following is the Rascal version of what was inside the for loop in the example Go code:

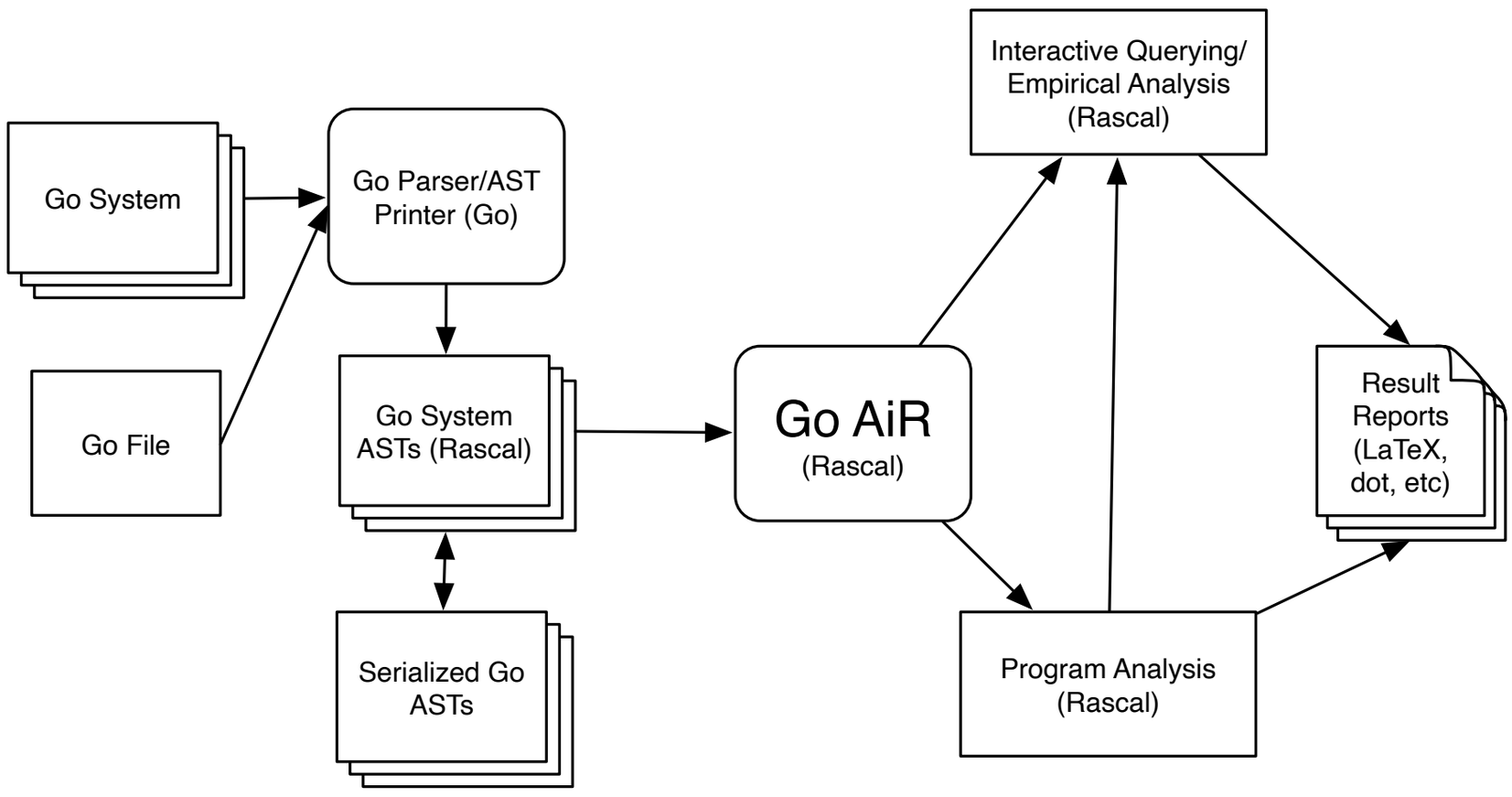
```
if (valueSpec(names,someExpr(selectorExpr(ident("sync"),"WaitGroup")),_) := d) {
  featureDecls = featureDecls
    + { < d.at, featureDecl(d.at, n, waitGroupDecl())> | n <- names };
  }
}
```

- Pattern matching gives us a natural way to work with AST terms, built-in relation types and comprehensions help us with fact extraction and analysis

Image from: https://medium.com/nerd-for-tech/learn-golang-in-one-blog-fdd568e6f631

Go AiR

- Go AiR (Analysis in Rascal) is a prototype analysis framework for Go



What can we currently do?

- We can extract ASTs from Go source code (using a Go program to do this) and read them into Rascal, either for individual files or entire systems (tested across a large number of popular systems)

- We can serialize/deserialize these systems, along with additional extracted data

- We can explore Go code using Rascal's pattern matching features

- We can work with multiple releases of a system, based on Git version history

- We are moving earlier fact extraction code, written in Go, over to Rascal

What would we like to do?

- We want to redo our earlier work on traditional concurrency features and compare this to earlier work on message passing

- We want to integrate this with a rewriting logic semantics of Go, focused on concurrency, for concurrency analysis and verification

- We want to extract models of concurrent behavior to help developers understand the possible behaviors of their code

And now for some controversy

- We want to extend this to be interprocedural,
  but: for a really dynamic language, where even the decision of what code to include is deferred until runtime, is this even useful?

- To borrow from earlier: keep it simple! Do we even need to support the entire language for this to be useful for developers?

- For artifacts, are full VMs at all useful? Should we aim at using something like Docker? Images available in the cloud? Something else?

Discussion

Thank you!
Any Questions?

- Go AiR: https://github.com/PLSE-Lab/go-analysis
- Rascal: https://www.rascal-mpl.org/
- Me: https://cs.appstate.edu/hillsma/

Appalachian STATE UNIVERSITY