A Plague of Gofers:

Generalized Rule-Based Data Entry With Lazy Data Retrieval

William H. Duquette
Jet Propulsion Laboratory, California Institute of Technology
William.H.Duquette@jpl.nasa.gov

Abstract

A gofer value is a data value that tells the application how to retrieve a desired piece of data on demand, according to some rule, and a gofer type is the code that validates gofer values and retrieves the data on demand. The advantage of using a gofer is that the value returned can change over the lifetime of the gofer value as state of the application changes. The gofer infrastructure allows the definition of gofer types consisting of many different rules, with support for GUI creation and editing of gofer values.

For example, the user may desire that a particular simulation input affect all civilian groups residing in a particular set of neighborhoods. Instead of listing the groups explicitly, the user chooses the relevant rule and the neighborhoods of interest; at each time step, the simulation can determine the groups that currently reside in the chosen neighborhoods.

1. Background

The Athena Regional Stability Simulation is a model of political actors and the effects of their actions in a particular region of the world. The region is divided into neighborhoods, in which reside various civilian groups. The actors have a variety of assets, including money, military and police forces, and means of communication, which they use to achieve their political ends. The user of Athena models the behavior of the actors in the simulation by defining their strategies, which consist of a prioritized list of tactics; then, the actor's behavior is determined by the tactics the actor executes.

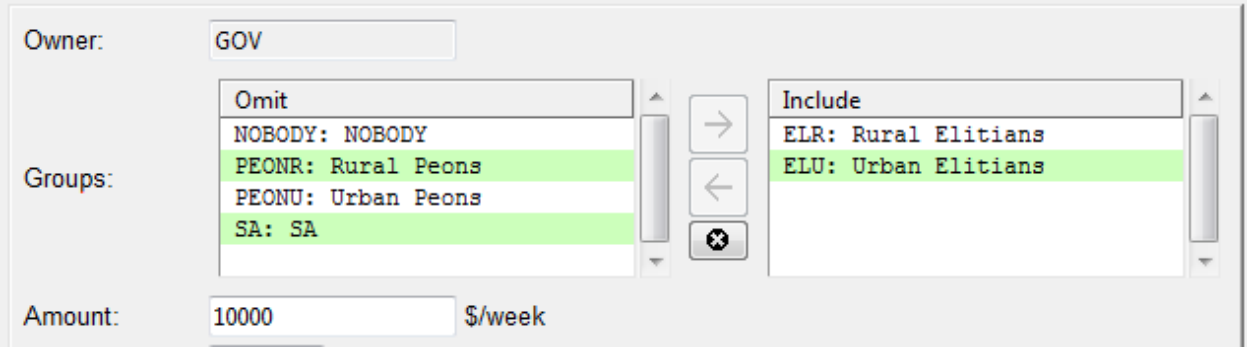
2. The Problem

Consider a tactic that performs a particular bit of behavior: for example, paying money to fund social services for particular groups in the civilian population. The tactic has several parameters, of which these are the most important:

- The list of groups for which services are to be funded

- The amount of money to spend during the current week

Now, there are any number of ways the user might select the list of groups and the sum of money. He might want the groups that reside in a particular neighborhood, or the groups that support the actor who will execute the tactic. He might want the actor to spend a specific amount of money, or the amount of money required to achieve a particular result, or 40% of the actor's available cash. From a software point of view, however, we have always required that the user enter the precise list of groups and the precise sum of money.



Now suppose the user wants to select all civilian groups that support a particular actor. He has to go through the output data to figure out which groups those are, and enter them in the tactic's dialog. If there are many of them, this is tedious and error-prone. And worse than that, tactics are defined before the simulation begins to run, but the list of groups that support an actor can change as the simulation is running. If he enters a specific list of groups, it may become incorrect over the course of the run. In short, there is really no way to do what the Athena user wants to do.

3. The Solution: Rule-based Data Retrieval

The solution is to redefine the tactic parameters. Instead of entering the desired data values, the user enters rules for retrieving the desired data values. That is, instead of

- The list of groups for which services are to be funded
- The amount of money to spend during the current week

the user enters

- A rule for selecting a list of civilian groups
- A rule for choosing a sum of money

When it is time to execute the tactic, the tactic code evaluates the rules to retrieve the desired data values.

This solves the problem nicely, but it poses two further challenges. First, we need to be able to edit these rule descriptions in the GUI in a user-friendly way. Second, there is likely to be a large number of these data-retrieval rules for any given data type (i.e., "list of civilian groups"). The code related to any particular rule needs to be both concise and maintainable. Athena's "gofer" concept handles both of these challenges.

A *gofer type* is a collection of rules for retrieving data of a particular type (e.g., lists of civilian groups). The rules are referred to as *gofer rules*. A *gofer value*, also called a *gdict*, is a dictionary that specifies the gofer type, the specific rule, and any additional data required by the rule (e.g., a list of neighborhoods). Given a gofer value, we *evaluate* it to retrieve the relevant data.

The gofer infrastructure provides tools for simply and concisely building gofer types out of rules. In addition, each gofer type is associated with a dynaform [1] that can be used to create and edit values of the type within the GUI.

4. Gofer Types

A gofer type is a type-definition object [2] that collects together a set of related gofer rules. That is, it is an ensemble whose subcommands operate on gofer values that belonging to the type. Like all type-definition objects it includes a `validate` method, of which more later; but more importantly it includes an `eval` method, which is used to evaluate gofer values and retrieve the desired data.

4.1 Evaluating Gofer Values

For example, suppose the user wants to select all civilian groups resident in neighborhoods N1 and N2. This is an application of the `gofer::CIVGROUPS` gofer type, which returns lists of civilian groups. The corresponding gdict looks like this:

```
% set gdict {_type CIVGROUPS _rule RESIDENT_IN nlist {N1 N2}}
```

Every gdict has a `_type` key and a `_rule` key, along with keys for any rule-specific parameters. In this case, the `nlist` parameter is a list of neighborhoods. Evaluating this value will return a list of the names of groups resident in the two neighborhoods:

```
% gofer::CIVGROUPS eval $gdict
G1 G2 G3 G4
%
```

4.2 Validating Gofer Values

The gofer type's `validate` method takes a gdict and validates it, throwing an error with error code INVALID if any problem is found and returning the gdict in canonical form otherwise.

Gofer values derive from user input, and Athena tends to be forgiving of user input. Group names, for example, are canonically in upper case, but Athena allows group names to be entered in lower case as well. Thus, a validate method is responsible not only for finding errors, but for putting values into the form the application expects.

In the case of a gdict, the `_type` and `_rule` keys are canonically the first two keys, and their values are canonically in upper case. The canonical form of other keys is naturally rule-dependent.

For example:

```
% set gdict {_type civgroups _rule resident_in nlist {n1 n2}}
% gofer::CIVGROUPS validate $gdict
_type CIVGROUPS _rule RESIDENT_IN nlist {N1 N2}
```

4.3 Generating Narrative Strings

It is often desirable to display a gofer value in the GUI, but the standard gdict isn't terribly readable for the average user. The gofer type's `narrative` method takes a gdict and produces a human-readable narrative string, suitable for embedding in a longer sentence. For example,

```
% set gdict {_type CIVGROUPS _rule RESIDENT_IN nlist {N1 N2}}
% gofer::CIVGROUPS narrative $gdict
all civilian groups resident in neighborhoods N1 and N2
```

4.4 Representing Raw Inputs

Of course, sometimes one will want to choose the list of groups by hand, in the old-fashioned way. Consequently, every gofer type has a `BY_VALUE` rule with one parameter, `raw_value`, that is used to represent a value chosen by hand. For example,

```
% set gdict {_type CIVGROUPS _rule BY_VALUE raw_value {G1 G4}}
```

This gdict simply evaluates to its raw value:

```
% gofer::CIVGROUPS eval $gdict
G1 G4
```

4.5 Auto-Translation of Raw Inputs

To ease the integration of gofers and the import of older Athena scenarios after gofers have been added, we allow for the following special case. On validation, if a gofer value does not begin with the `_type` key we assume that it is simply a raw value, and translate it into a gdict with the `BY_VALUE` rule. For example,

```
% gofer::CIVGROUPS validate {G1 G2 G3}
_type CIVGROUPS _rule BY_VALUE raw_value {G1 G2 G3}
```

Of course, the `raw_value` must also be a valid value for the gofer type's `BY_VALUE` rule.

5. Defining a Gofer Type

At base, a gofer type is simply an ensemble command with the right subcommands and semantics; it could be implemented as a namespace ensemble, or as a Snit type ensemble [3], and we started with the latter. It developed that distinct gofer types have a great deal of mechanism in common, and so the type ensembles evolved into instances of a Snit type called `goferType`. Even then, there was a boilerplate code that needed to be written over and over for each type.

Consequently, gofer types are created using the `gofer define` command, which creates the instance of goferType and also performs a number of other housekeeping chores. Then, once the type object exists, rules are added to it.

5.1 Creating the Gofer Type

For example, the `gofer::CIVGROUPS` type is created as follows:

```
gofer define CIVGROUPS {
    rc "" -width 3in -span 3
    label {
        Enter a rule for selecting a set of civilian groups:
    }
    rc

    rc
    selector _rule {
        case BY_VALUE "By name" {
            rc "Select groups from the following list:"
            rc
            enumlist raw_value -dictcmd {::civgroup names} \
                -width 30 -height 10
        }
        . . .
    }
}
```

First, this command creates an instance of `goferType` called `::gofer::CIVGROUPS`, and registers it with the `gofer` command.

Next, it specifies a dynaform to use for editing values of the gofer type. The first field in the dynaform script must be a `selector` field called `_rule`, with one case for each of the type's rules. The `case` names must match the rule names.

Every gofer value includes a `_type` key, and yet no `_type` field appears in the script. This is because `_type` has to appear at the beginning of every gofer type's dynaform script as an invisible context field, and so `gofer define` adds it in automatically.

5.2 Adding a Rule to a Gofer Type

Initially, the new type will have no rules associated with it; they must be implemented individually. Each rule is represented in the code as a type-definition object for the rule's own parameters in the gdict. Again, this rule object could be implemented as a namespace ensemble or a Snit type ensemble; but as before we discovered that distinct rule objects had a certain amount of boilerplate code in common. For convenience, then, rule objects are defined using the `gofer rule` command, which takes a partial Snit type definition script, adds boilerplate, and registers the rule with its gofer type object.

For example, here is the definition of the `BY_VALUE` rule:

```
gofer rule CIVGROUPS BY_VALUE {raw_value} {
    typemethod validate {gdict} {
        dict with gdict {}
        dict create raw_value \
            [listval "groups" {civgroup validate} $raw_value]
    }

    typemethod narrative {gdict {opt ""}} {
        dict with gdict {}
        return [listnar "group" "these groups" $raw_value $opt]
    }

    typemethod eval {gdict} {
        dict get $gdict raw_value
    }
}
```

The `gofer rule` command takes four arguments:

- The type name, e.g., `CIVGROUPS`
- The rule name, e.g., `BY_VALUE`
- A list of the names of the rule's parameters, e.g., `{raw_value}`
- A Snit type body with type methods for the essential rule operations: `validate`, `narrative`, and `eval`.

It creates a rule object, a Snit type ensemble, called `::gofer::CIVGROUPS::BY_VALUE`, and registers it with the `CIVGROUPS` type.

5.3 Semantics of Gofer Rule Operations

The rule operations differ slightly from the similarly named gofer operations.

- The `validate` operation takes a gdict and validates only the keys that are associated with this particular rule. Any other keys are ignored. It returns a dictionary containing only keys associated with this particular rule, with the values in canonical form.

- The `narrative` operation returns a narrative string given a valid gdict for this rule. As with `validate`, it ignores any keys but those associated with this rule, e.g., it ignores the `_type` and `_rule` keys.

- The `eval` operation evaluates the gdict; again, it ignores any keys but those associated with this rule.

5.4 Helper Commands

One of the goals of the gofer system is that rule definitions should be as concise as possible; consequently, shared code has been ruthlessly abstracted. For example, the `validate` and `narrative` methods shown above for the `BY_VALUE` rule make use of the helper routines `listval` and `listnar`. The precise semantics of these commands doesn't matter for the purposes of this discussion; the main point is that they perform part of the job in a standard way.

To make adding helpers easier, the `gofer rule` command adds a namespace path containing `::gofer` and `::gofer::`*`typename`* to the rule object's namespace. Thus, the `::gofer` module can provide helpers for use by any rule, and a given gofer type can provide helpers for use by its own rules, simply by defining procs within their namespaces.

5.5 Sharing Rules

It is not uncommon for a rule to be used by more than one gofer type. For example, Athena has three kinds of group, and so in addition to `gofer::CIVGROUPS` we have also defined `gofer::GROUPS`, which returns a list of any kind of group. But a list of civilian groups is also simply a list of groups, and so almost all of the rules in `gofer::CIVGROUPS` can shared with `gofer::GROUPS`. This is done using the `gofer rulefrom` command.

For example, the following command is part of the definition of gofer::GROUPS:

```
gofer rulefrom GROUPS CIV_RESIDENT_IN \
    ::gofer::CIVGROUPS::RESIDENT_IN
```

The `gofer::GROUPS`' `CIV_RESIDENT_IN` rule is simply mapped to the given rule object.

5.6 Gofer Rules and Dynaform Cases

As noted in Section 5.1, a gofer type has a dynaform that has a selector case for each of the type's rules. It might seem like each rule's case script could be defined by the rule object, and the full dynaform built up from the pieces. The difficulty is that the descriptive text in the dynaform case often needs to be slightly different for each use of the rule, depending on the type with

which it is associated. Rather than imposing an unpleasant consistency across the GUI for all types that use a rule, we chose to leave the dynaform script in one piece.

5.7 Invoking a Gofer Operation

The gofer type and its rule objects work together to implement the three primary gofer operations: `validate`, `narrative`, and `eval`. When the operation is passed a gdict, the gofer type determines the rule from the `_rule` key, and passes the gdict along to the rule object, returning the result.

Only the `validate` operation involves any additional complexity. The rule object's `validate` method only validates and canonicalizes the rule-specific parameters. The gofer type's validate method appends the result to a stub containing the `_type` and `_rule` keys, and returns that.

5.8 Sanity Checking

A gofer type is built out of many small pieces, all of which need to hook together just right. In particular, each of a gofer type's rules has to have a case in the type's dynaform (and vice versa), and the case has to contain a field for each of the rule's parameters.

One could wait for the user to discover any mismatches; instead, we wrote a routine, the `gofer check` command, which does a complete sweep of all defined gofer types and throws an error if any problems are found. This routine is called by the application's test suite, ensuring that all potential problems are found as part of the build process.

6. The `gofer` Convenience Command and the `_type` Key

As noted frequently above, each valid gdict begins with the `_type` key, which names the specific gofer type. One might think that since gofer types are distinct in use—if you need a list of civilian groups, you don't use a gofer type that retrieves quantities of money—that this piece of information is extraneous. And in fact, our initial implementation omitted it.

However, Athena is scriptable; and since there are user commands for creating tactics, we also need user commands for constructing and evaluating gofer values. For convenience, then, we added the `_type` key to support `gofer` subcommands that operate on gdicts of arbitrary type.

For example,

```
% set gdict [gofer construct civgroups resident_in {n1 n2}]
_type CIVGROUPS _rule RESIDENT_IN nlist {N1 N2}
% gofer narrative $gdict
all civilian groups resident in neighborhoods N1 and N2
% gofer eval $gdict
G1 G2 G3 G4
%
```

7. Status and Future Work

Gofers are a new feature in Athena. At present, we have defined four gopher types, for lists of actors, lists of civilian groups, lists of force groups, and lists of groups in general. The total number of rule objects is currently around 40, with `gofer::GROUPS` defining a number of rules of its own and reusing most of the `gofer::CIVGROUPS` and `gofer::FRCGROUPS` rules. These gofers are currently used by only two of Athena's eighteen tactics.

Clearly there is much work left to be done. We need to update the existing tactics to use gofer parameters as appropriate. As we do so, we will discover additional gofers that we need to define. And then, the sets of rules included in the four existing gofers are only preliminary; as the users become more familiar with gofers and what they can do, they will have their own ideas as to what would be useful. It's early days yet.

Nevertheless, the gofer architecture appears to be up to the job. New gofer types can be added without disturbing existing gofer types, and new rules can be added to any existing gofer type without disturbing other rules, or breaking any existing scenario data that uses the existing rules. In that sense, the gofer system is complete; it now simply remains to exploit it ruthlessly.

8. References

[1] Duquette, William H., "Dynaforms and Dynaviews", 20th Tcl/Tk Conference, Proceedings.

[2] Duquette, William H., "Type-Definition Objects", 12th Tcl/Tk Conference, http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/37573/1/05-2409.pdf.

[3] Duquette, William, Snit Object Framework, found in Tcllib, http://tcllib.sourceforge.net/doc/snit.html.

9. Acknowledgements

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, during

the development of the Athena Stability & Recovery Operations Simulation (Athena) for the TRADOC G2 Intelligence Support Activity (TRISA) at Fort Leavenworth, Kansas.