FSFrame
Simple Frame System

Mihai Maruseac
IA, 2011

1. Description

The FSFrame project implements a framework to work with knowledge represented using a Frame Representation. Basically, each knowledge atom is represented either as a frame or a frame slot. A frame is a list of slots, each of them being something like an attribute/value pair. In fact, a slot can also have a default value and two actions: one to be executed when the value of the slot is required and one to be executed when the value of the slot is updated.

Each frame has a parent frame, the only prebuilt frame being ROOT. Each frame inherits all slots from it's parent. When a slot value is required from a frame it is searched there. If it is not found, the search continues to the parent or to the default value or the if-needed action is called. These search modes are controlled via user preferences.

The if-added action of a frame is also executed when a descendant of this frame is the target of an add action from the user, when a new slot is added. This action can be used to compute some statistics or to elaborate some new pieces of knowledge.

2. Features

Aside from the above mentioned features, the FSFrame project allows the user to have an overview of the entire Frame System at any point using diagrams. This can be achieved via a user command, see below in section 4 the exact syntax.

Moreover, each action can be user defined. The FSFrame project implements a simple interpreter for a subset of expressions. Sections 3 and 4 describe this.

3. Design

The initial design wanted to allow for a GUI application as well as a TUI one. The GUI should provide a point-and-click interface to frame manipulation as well as an interpreter for user commands while the TUI only functioned as an interpreter.

However, due to lack of time, the GUI was deferred to a later release. To add it, the developer only has to implement functions calling functions from Frame.TUI module and change the main function of the application to dispatch to the GUI main loop. Basically, two changes are enough to switch the application from a TUI based one to a GUI based one – provided that the GUI stubs are implemented already.

As for the TUI aspect of the application, it is a simple REPL (Read-Eval-Print-Loop). The user is expected to provide some commands, as described in section 4 and the application will try to execute them. Since the syntax is a little bit too rigid, there could be a high number of parse errors but the code treats them gracefully and the application is not killed.

The user can define his own actions for the two action values of a slot. Although the interpreter is very simple, an action can be used to build strings, to compute new values or to test some conditions. Future improvements will allow a complete interpreter, making the action language be Turing complete.

Aside from what was required and already describedI have implemented a functionality to execute multiple commands stored into a single file just to speed up the initial filling of the world. A later feature, extending on this one, would be to implement proper save and load of the entire frame description but this too was deferred.

Last but not least, the application allows the user to see a graphical description of the world at any point in the REPL loop. By issuing a simple command a file with the world description is created and that image is presented to the user. It is possible to construct several such files and analyze them offline, even after the application was closed.

When the GUI interface is created, the graphical representation can be embedded into it by using a single IO call to load an image. Thus, it is very easy to add the GUI in the next iteration of this product.

4. Description of user commands

The entire syntax of user commands is defined by Haskell datatypes in Frame/Types.hs and rephrased here for ease of understanding. Adding a new command reduces to changing this file and adding a new case to a function in Frame/FrameOps.

There are two kinds of user commands. Commands which can be issued only from the REPL loop and commands which can be issued from scripts. There is no special syntax to distinguish between them because that would make command syntax look scarier.

```
usercmd = “QUIT”
        | “RUN” filename
        | “DUMP”
        | “GRAPH” filename
        | “EXEC” cmd
        | “EVAL” expr
```

The first four commands presented above can only be used from REPL, using them from a batch-script or an action will print an error and will abort current execution.

```
cmd = “FCREATE” frame_name frame_name frame_type
    | “FPUT” frame_name slot_name put_type
    | “FSETPARAMS” param_setting bool_value
```

These commands allow the user to change the knowledge base adding new frames or new slots or changing preferences concerning actions and default values.

```
put_type = “PutV” obj
         | “PutD” obj
         | “PutVE” expr
         | “PutDE” expr
         | “PutN” action
         | “PutA” action
```

The `put_type` construct allows the user to build a slot without fully specifying it's content. It is enough to only provide what you are modifying, the application will take care of the rest. Also, there are two `put_type` values which receive an expression as an argument. This allows user to put values depending on the result of some expressions built by them.

```
param_setting = DefaultsEnabled
              | ActionsEnabled
              | DefaultsThenNeeded
              | SearchTypeIsZ
```

The values represent the only parameters that the user can change in this application. They set whether default values are used when getting the value of a slot, whether actions are enabled (if actions are triggered while disabled an error is raised), whether the default values are consulted before triggering if-needed action and whether to give priority to slot's fields over parent's (Z search) or to slot value type over field (N search).

```
expr = “DOT” frame_name slot_name
     | “PREF” param_setting
     | “OBJ” obj
     | “FNAME”
     | “SNAME”
     | “SVAL”
     | “NOT” expr
     | “AND” expr expr
     | “OR” expr expr
     | “CONCAT” expr expr
     | “ADD” expr expr
     | “SUB” expr expr
     | “MUL” expr expr
     | “DIV” expr expr
```

The first production of the above rules allows an expression to be the value of a slot from a frame. In the background, it involves getting a parameter using one of the search methods described by `param_setting`. The second production allows the retrieval of a parameter setting while the third production allows evaluation of constant values.

The `“FNAME” “SNAME”` and `“SVAL” expr` productions are used as placeholders for the three action specific arguments: the frame and the slot triggering the action execution and the added value if the action is of if-added type. The application takes care of filling in those values. It is an error to use these values in other contexts.

The rest of the productions are expressions working with boolean values, with string values and with numbers. It is an error to use a construction which makes no logical sense (for example adding a number and a boolean value).

```
action = '[' usercmd (',' usercmd)+ ']'
```

An action is just a list of user commands, each of them being separated by commas. The list is enclosed in square brackets, just like a normal Haskell list. An action cannot be empty, it is an error to provide an empty list.

```
obj = R number
    | S any_string
    | B bool_value
number = any_valid_real_of_integer_number
filename = '”' any_string '”'
frame_name = frame_slot_string
slot_name = frame_slot_string
frame_slot_string = [a-zA-Z][a-zA-Z0-9_]*
bool_value = True | False
```

Remember that each string must be enclosed in quotes. Also, if a production has more terms in the RHS each apparition of the LHS which will lead to that expansion must be enclosed in parentheses, just like nested lists in Scheme or nested function application in functional programming languages.

5. Future improvements

While this application has a full range of features, both required and extra, there are more improvements which can be made. First of all, a GUI interface is desired. Luckily, it is very easy to add it, the problem is that there was not enough time to implement it in this version (because of author's lack of proper time management).

Another feature will be to relax the syntax of user commands, making them be closer to a natural language than they are right now. Until then, the user has to be very careful to write proper parentheses and don't include too much whitespace since this can make parsing fail. Still on the same subject, maybe adding libreadline support will make the TUI much more usable in the future.

Other improvement direction will be to implement proper save and load commands. Also, adding a command to do inferences and updating the world in real time are nice things to implement in the future.

But the most important feature to be implemented is extending the action interpreter such that the action language can become Turing complete allowing for arbitrary code to be executed.