New Ideas and Emerging Results Track: Towards Unifying, Functional Programing, Rule-Based Programming, and Event-Based Programming

Renaud Pawlak
LISITE-ISEP
28 rue Notre-Dame des Champs
Paris, France
renaud.pawlak@isep.fr

ABSTRACT

In this paper, we discuss rule-based programming with a new language called INI, which we created to evaluate rule-based oriented programming in terms of code simplicity, quality, and maintainability. INI takes inspiration from existing rule-based languages, mostly from languages permitting rewriting rules, but also from functional languages allowing imperative style programming. We shortly and informally present the INI syntax and semantics, and, using some simple examples, we show how to use INI for rule-based programming. We discuss rule-oriented code and we show that INI also allows to fall into functional programming style in a consistent way. Our claim, and the novelty here, is that rule-based programming mixed with functional programming here leads to more maintainable code, since it forces the programmers to explicitly write the pre-condition upon which some behavior shall be executed. In addition, it simplifies the code by replacing most imperative control-flow constructs such as various looping constructs and if then else constructs, with a consistent and unique rule construct. Last but not least, we show that our programming model allows for consistent integration of event-based programming through event-triggered rules. This event-based model allows for intuitive multithreaded programming, which makes it suitable for many types of applications such as embedded application and server applications.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Rule-based Programming

Keywords

Programming Language, Code Quality, Rule-based Programming, Functional Programming

1. INTRODUCTION

In this paper, we present INI, a rule-based programming language. In INI, all functions are defined with rules, i.e. guarded actions. An action is executed only if its guard is evaluated to true in the context of the function. Also, an action will continue to be evaluated while its guard is true. This specificity can be found in rewriting-based languages [...]. However, INI is not intended to deal only with structure rewriting, since the actions may have classical behaviors and side-effects, like in typical imperative languages.

Thus, INI allows rule-based programming style, which will be studied in this paper. First, we will see that rule-based programming allows for some algorithms to be implemented quite straightforwardly, and that it enforces the programmer to declare pre-conditions, thus leading to better programming style and code safety. Second, we will see that rule-based programming leads to straightforward asynchronous and multithreaded programming, thanks to event-triggered rules. Third, we will show that rule-based programming is not incompatible with functional programing, since INI allows the programmer to fall back on this style of programming when rule-based programming is not well adapted to the problem to solve. Finally, we will shortly present the INI implementation, which is built on Java and allows easy extension, through bindings, but also with the definition of new built-in functions and events. A small case study will also be discussed.

2. INI IN A NUTSHELL

In order to discuss our programming model, we first present INI very shortly. A complete reference manual is available at [...]

2.1 Overview

INI is a programming language where all functions are defined with rules. A rule is defined with a guard and an action. The action will be executed only if the guard is evaluated to `true` in the context of the function. In other words: when the guard is passed, it triggers the evaluation of the action. The action is a list of sequential statements including variable assignments, function invocations, and return

statements. These statements are similar to imperative language ones, except that they do not allow any control flow language constructs such as *ifs* or *loops*. In INI, a function does not terminate before all its rules' guards are evaluated to `false`, except when an action explicitly forces the function to end (with a return statement or an error).

Syntactically, a function is written as:

```
function <name>(<parameters>) {
  <guard1> { <statements> }
  <guard2> { <statements> }
  ...
  <guardN> { <statements> }
}
```

Guards are either an event expression, or a logical expression, or both:

```
<guard> := <logical_expr>
         | <event_expression>
         | <event_expression> <logical_expr>
```

Logical expressions are formed of variable accesses, function invocations, and literals (strings, numbers, ...), composed together with classical logical operators (`&&` (and), `||` (or), `!` (not)) and/or comparison operators (`==`, `!=`, `>`, `>=`, `<`, `<=`). INI also defines a match operator ($\sim$) that allows for regular expression matching on strings and for type matching, similarly to the match construct in OCAML. In guards, logical expression must be evaluated to `true` to allow the evaluation of the rule's action. If a logical expression is not specified (event-only expression), it is equivalent to an always-true expression.

Event expressions are more original since they allow the programmer to specify that the rule evaluation is triggered upon one or several events. Events start with `@`, have a name, and can take parameters. In an event expression, events can be composed with two main temporal operators: sequentiality (`;`) and co-incidence (`&`). The expression `@a(..);@b(..)` means that the `@a` and `@b` events must happen in that order, while `@a(..) @b(..)` means that `@a()` and `@b()` must happen in any order. `@a(..)&@b(..)` means that both event must happen at the same time, which implies that at least one of the events has a duration. When a guard contains an event expression, it means that the rule is event-trigerred and is evaluated asynchronously whenever the event or events occur in the way specified by the expression. INI provides two special events for function initialization and termination: `@init`, which occurs when the function evaluation starts, i.e. before any other events or rules, and `@end`, which occurs once the function ends, that is to say when no rules can apply anymore (all the guards are evaluated to `false`). For a given function, `@init` and `@end`-triggered rules are executed only once and take no parameters.

2.2 Example

To illustrate INI, let us show a simple example that implements some sort of infinite ping-pong game between two rules.

```
1  function ping_pong() {
2    @init() {
3      v=1
4    }
5    v == 2 {
6      println("pong")
7      v = 1
8    }
9    v == 1 {
10     println("ping")
11     v = 2
12   }
13 }
```

When entering `ping_pong`, INI first evaluates the `@init`-trigerred rule at line 2 that initializes a `v` variable to 1. Then, INI sequentially tries out all the rules of the function in their declaration order. Here, the guard at line 5 evaluates to `false`, but the one at line 9 evaluates to `true`, thus triggering the evaluation of the action that prints out `"ping"` (line 10) and sets `v` to 2 (line 11). After that, remember that the function will not terminate before all the guards evaluate to `false`. So, since line 5 now evaluates to `true`, INI evaluates the rule that prints out `"pong"` (line 6) and resets `v` to 1 (line 7). This action obviously triggers again the other rule, thus starting the endless ping-pong game over again. Note that the rules declaration order is not important here. Any other rule order will give the same result. That's because the guards are *disjoint*.

3. RULE-BASED PROGRAMMING

3.1 Programming the Behavior

From a behavioral perspective, what is interesting about the Rule-Based Paradigm is that it clearly shows the conditions upon which some code can be executed. Indeed, guards can be seen as precondition for the actions. Because of this, the code representation naturally shows the states and transitions of the functions, if any. For instance, if we take again the `ping_pong` function, it is pretty straightforward to see that it corresponds to the following state machine:

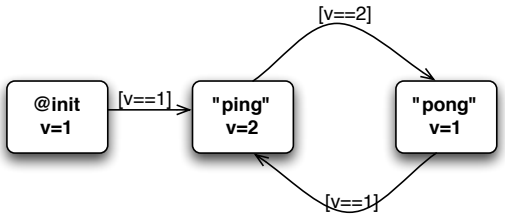

Figure 1: The `ping_pong` function state machine

To show are more complicated example, the following code reads some text from the keyboard and writes `"hello"` to a file named after the read text. Also, the file is not written if it already exists on the file system.

```
1  function write_hello() {
2    !fname {
3      print("enter a file name> ")
4      fname = read_keyboard()
5      f_exists = file_exists(fname)
6    }
7    fname && !f_exists {
8      fprint(fname,"hello")
9      clear(fname)
10   }
11   fname && f_exists {
12     println("'"+fname+"' already exists")
13     clear(fname)
14   }
```

```
15 }
```

The three rules at lines 2, 7, and 11 show all the possible transitions depending on the variables states. Note that `!fname` is `true` if `fname` is undefined, `false` otherwise: within a guard, a non-boolean variable evaluates to a boolean value depending on its definition state (defined/undefined). Note also the use of the `clear(fname)` built-in functions at lines 9 and 13 that reset the `fname` variable and go back to the initial rule of line 2. Finally, an analysis of the function allows us to build the following state machine, which shows the possible transitions.

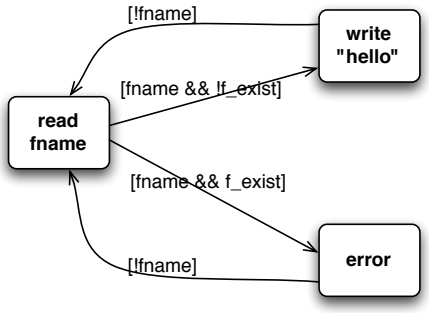

Figure 2: The `write_hello` function state machine

Similarly to the `ping_pong` function, the `write_hello` function never terminates. This may seem ackward to programmers used to functional programming, where non-termination means infinite recursion. It is however completely normal behavior for interactive programs such as servers, data exchange protocols, batch jobs, etc. In such kinds of programs, it is actually important that the program never terminates and that the entire state space (possibilities) is clearly handled by the program. Our first experiments with rule-based programming show that the code for such interactive programs looks simple and intuitive to understand.

3.2 Asynchronism and multithreading

Most interactive programs require the use of asynchronous events and parallelism. For instance, a batch job may need to perform a given task at a given time of the day, a server will require several clients to be served in parallel, and so on. For such cases, INI allows for event-triggered rules, which enable event-based programming with implicit parallelism. Such event triggering is also found in some specific languages used for programming complex and highly interactive systems, such as languages used in video games and robotics, for example Urbiscript [...].

The advantage of INI over other event-based languages, is that its rule-based programming model makes it possible to cleanly and intuitively integrate events as part of the guards. Example of INI built-in events are `@every(time)`, which is triggered at regular time intervals given in milliseconds, `@cron("crontab expression")`, which allows for the scheduling of events at specific time and dates (using an underlying crontab engine [...]), `@error(message)`, which is triggered when an error occurs or is thrown by the program, and last but not least, `@update(var1, var2, ... varN)`, which is triggered when one of the given variables is updated by the program (with an assignment).

Using the `@update` event, we have programmed an HTTP server prototype with INI. It uses the socket API. The `socket_accept(s)` at line 6 is a blocking call that waits until a client connects to the server socket created at line 3. At this point, instead of dealing with the connection right after the `socket_accept(s)` call returns, we simply set the returned socket to the `c` variable, that we observe using the `@update(c)` event at line 9.

```
1  function start_http_server(port, handler) {
2    @init() {
3      s = socket_server(port)
4      println("Server started on "+port)
5    }
6    s {
7      c = socket_accept(s)
8    }
9    @update(c) {
10     // handle http requests here
11     client = socket_address(c)
12     ...
13   }
14 }
```

INI events are executed in separate thread, within the function context, thus enabling implicit parallelism. This way, our HTTP server can handle several clients in parallel. Using this technique, we have also programmed a prototype of an M2M gateway that reads some sensors, stores the value in some files, and every day at midnight (`@cron("0 0 * * *")` event), it creates a zip and sends it by FTP to a server.

4. FUNCTIONAL PROGRAMMING

As explained before, rule-based programming is well suited to program highly interactive systems. However, it is sometimes more appropriate to use a functional-programming style, especially when a function needs to return the result of a calculation, and even more when this calculation is easy to express in a recursive manner.

4.1 Functional v.s. Rule-Based Style

The INI programming model makes it possible to switch from a rule-based paradigm to a functional paradigm so that both styles can consistently cohabit within the same program. For instance, the following code shows both versions of a factorial function.

```
function fac(n) {
  @init() {
    f=1
    i=1
  }
  i <= n {
    f=f*i++
  }
  @end() {
    return f
  }
}
```

```
function fac(n) {
  n==1 {
    return 1
  }
  n > 1 {
    return n*fac(n-1)
  }
}
```

On the left, we can see the rule-based version, that defines 3 rules:

1. an initialization `@init`-triggered rule that defines and initializes a variable to store the result (`f`) and a variable to store the current integer to multiply (`i`).
2. a rule that multiplies the result by the current integer `i` and increments it, guarded with the condition that `i` is

lower or equal to the number `n` we want to calculate the factorial of. Due to the INI execution semantics, this rule will continue to apply until `i<=n` becomes `false`, which eventually happens since `i` is incremented at each rule execution (`i++` expression). Once `i` equals to `n`, there are no rules to be executed anymore in the function.

3. a termination `@end`-triggered rule that returns the calculated result `f`.

The functional version on the right-hand side is quite simpler since factorial calculation is easy to define recursively. Thus, we can define one rule that recursively uses `fac(n-1)` to calculate `fac(n)` for `n>1`, and a non-recursive rule for `n==1`, that simply returns 1 (the factorial value for `fac(1)`) for terminating the recursion. Note that each rule causes the function to end immediately because of the `return` statement.

Similarly to imperative style v.s. functional style, both rule-based and functional styles have advantages and drawbacks depending on what needs to be done. So, we think that it is good that both styles consistently cohabit within the same programming model.

4.2 Type Matching

In order to fully allow functional programming style, it is import to enable ways to construct algebraic types, that allow the programmers to define complex structures. For the program to be well-typed when using these structures, INI supports a match operator, which is similar to the match construct in advanced functional languages such as CAML and Haskell. To demonstrate the use of such types, we show an INI program that allows the construction of algebraic expressions, and their evaluation with an `calc` function. This function uses the match operator within the rules guards to switch to the right action depending on the actual type constructor.

```
type Expr = Number[value:Float]
          | Plus[left:Expr,right:Expr]
          | Mult[left:Expr,right:Expr]
          | ...

function main() {
  @init() {
    // construct (3.0*2.0)+1
    expr = Plus[
      left=Mult[
        left=Number[value=3.0],
        right=Number[value=2.0]],
      right=Number[value=1.0]]
    println(expr+" = "+calc(expr))
  }
}

function calc(expr) {
  expr ~ Number[value] {
    return expr.value
  }
  expr ~ Plus[left,right] {
    return calc(expr.left)+calc(expr.right)
  }
  expr ~ Mult[left,right] {
    return calc(expr.left)*calc(expr.right)
  }
  ...
}
```

Note that INI also support other features coming from functional programming such as higher order functions. Although not proven, we do not see any reason yet why rule-based programming could not cohabit with functional programing. It is however not the main goal of INI to fully support functional programing. The next section relates much more important goals.

5. TYPING AND VALIDATIONS

In this section, we present the most important feature of INI: the typing, which is implemented with a type inference algorithm, and the validations, which is work in progress to statically check that the program is correct.

5.1 Motivations

Formal methods propose techniques to prove that the program behaves accordingly to some formal specifications. Formal methods can be categorized in proof-based frameworks, such as B and Coq, and model-checking tools, that try to cover the entire state-space to ensure some properties. The difficulties with formal methods is that they require the formal modeling of the programs, which is a hard and expensive task to achieve. Other validation techniques called static analysis techniques start from the program actual code to check that it is correct. Static analysis tools can be seen as extra compilation stages that go far beyond typing. Partial evaluation [...], control-flow analysis [...], symbolic evaluation (Javapath) [...], points-to analysis [...] are examples of static analysis techniques that can be used to automatically detect programming mistakes such as the use of undefined variables, out of bounds variables, unused variables and/or code.

Although theorically

5.2 Type inference

INI provides *type inference* [...], so that the programmer does not need to declare any type (except in some cases as we will see later). For instance, the `i=0` statement will define the `i` variable with an `Int` type. If the programmer tries to set the type of `i` to any other type within the `i` definition scope, (for instance with the `i=0.0` statement that assigns i with a `Float` type) the INI type checker will raise a type mismatch error. Following the same type inference principles, accessing a variable with the square brackets map access construct will automatically define the type of the variable to be a map. For instance, the `l[i]` expression tells INI that `l` is of type `Map(Int,T)` (i.e. a list of `T`), where `T` can be any type.

As said in Section **??**, most types in INI are calculated with the type inference engine, which allows the programmers to say very few about the typing. The kernel of this type inference is based on a Herbrand unification algorithm, as depicted by Robinson in [**?**]. The typing algorithm is enhanced with polymorphic function support, abstract data types (or algebraic types) support, and with internal subtyping for number types.

Thus, INI works with the following steps:

1. the parser constructs the AST;

2. an AST walker constructs the typing rules that should be fulfilled for each AST node and add them to a constraint list;

3. a unification algorithm is run on the type constraints, if conflicts are detected, they are added to the error list;

4. if errors are found, they are reported to the user and INI does not proceed to the execution phase.

5.3 Rule-based validations

6. CONCLUSION AND FUTURE WORK

In this paper, we discuss rule-based programming with a new language called INI, which we created to evaluate rule-based oriented programming in terms of code simplicity, quality, and maintainability. INI takes inspiration from existing rule-based languages, mostly from languages permitting rewriting rules, but also from functional languages allowing imperative style programming. We shortly and informally present the INI syntax and semantics, and, using some simple examples, we show how to use INI for rule-based programming. We discuss rule-oriented code and we show that INI also allows to fall into functional programming style in a consistent way. Our claim, and the novelty here, is that rule-based programming mixed with functional programming here leads to more maintainable code, since it forces the programmers to explicitly write the pre-condition upon which some behavior shall be executed. In addition, it simplifies the code by replacing most imperative control-flow constructs such as various looping constructs and if then else constructs, with a consistent and unique rule construct. Last but not least, we show that our programming model allows for consistent integration of event-based programming through event-triggered rules. This event-based model allows for intuitive multithreaded programming, which makes it suitable for many types of applications such as embedded application and server applications.