INI Language Reference Documentation / User Manual

Renaud Pawlak, CINCHEO

Version pre-alpha 2

Contents

1 Introduction

INI is a programming language dedicated to distributed computing. It natively handle processes, communication, deployment and synchronization. There are two kinds of first-class entities with INI: functions and processes. Functions are evaluated synchronously while processes are evaluated asynchronously and react to their environment through events. Functions return values. Processes communicate with each others through channels (like in π-calculus or in most multi-agent systems).

INI has been designed to remain as simple as possible for readability and maintainability, but also to ensure that it is easy to validate using formal methods such as Model Checking. Thus, a strong design choice is to allow only rules within the the definitions of functions and processes. A rule is a pair of (guard, action). The action is executed only if the guard is evaluated to true in the context of the function/process. Guards are formed of an optional event part (e.g. `@event(arguments) [configuration parameters]`), and of an optional condition part, which includes first-order logic expressions. At least one event part or a condition (possibly both) must be defined. 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. Besides, INI allows a `case` statement, which is also a list of rules.

Within a function or a process body, rules are executed until no rules are left to be executed. If one rule remain to be executed, the function or process does not terminate. On contrary to regular imperative languages, INI does not include any control structures such as *ifs* or *loops*. All the control is made through rules (i.e. guards and actions evaluation) and case statements (which are also rules).

For instance, a way to implement a factorial function `fac(n)` with INI would be through 3 rules:

1. an initialization 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 predicate that `i` is lower or equal to the number `n` we want to calculate the factorial of.

3. a termination rule that returns the calculated result `f`.

INI defines some default events that can be used in the guards of the rules. The first basic event is `@init`, which occurs when the function evaluation starts. The `@init` event is triggered before any other events or rules, even if not placed at the beginning of the function. The second basic event is `@end`, which occurs once the function ends, that is to say when no rules can apply anymore (all the guards evaluate to false). `@init` and `@end`-triggered rules are executed only once and take no parameters. Thus, the following code implements the factorial function in INI.

```
1  function fac(n) {
2    @init() {
3      f=1
4      i=1
5    }
6    i <= n {
7      f=f*i++
8    }
9    @end() {
10     return f
11   }
12 }
```

One can find the initialization and termination rules at lines 2 and 9. At line 6, the only non-event-triggered rule actually calculates the factorial result. It is important to understand that due to the INI execution semantics, the applicable rules will continue to apply until their guards evaluate to false. Here, `i<=n` eventually becomes false since `i++` increments `i` at each rule execution. Once `i` equals to `n`, there are no rules to be executed anymore in the function, and the `@end` event is triggered, thus making the function return the value stored in `f`.

2 Getting started with INI

2.1 Installing INI

The current implementation of INI is an interpreter written in pure Java, which means that INI is easy to install and run on any platform that supports the latest Java version. To install INI, go to `https://github.com/cincheo/ini` and follow the indications of the getting started section.

2.2 Write and run your first INI program

To write your first INI program, you can use your favorite editor. The convention is to name the source code files such as: `my_path/my_program.ini`, however, INI will not prevent the programmers to use their own file naming conventions at their own risks. Let us now write a simple "hello world" program. We just need a main process that prints out the text on the initialization event. Create a `hello_world.ini` file with the following content:

```
1 process main() {
2   @init() {
3     println("hello world")
4   }
5 }
```

To run this program, type in "`ini hello_world.ini`" in a console. The output should be "`hello world`".

You can now start using rule-based programming. For instance, the following program prints out "`hello world`" ten times.

```
1 process main() {
2   @init() {
3     i = 10
4   }
```

```
5   i > 0 {
6     println("hello world")
7     i--
8   }
9 }
```

Note that `println` is part of INI's built-in functions. For a list of these functions, see 3.2.5. Besides using built-in functions, it is possible to use Java objects from INI, as explained in 3.2.6.

3 Language features

3.1 Built-in types

INI comes with 5 built-in types for numbers: `Double`, `Float`, `Long`, `Int`, and `Byte`. They differ on the number of bytes used to encode them (same as Java encoding). INI allows the use of `Int` and `Float` literals: e.g.:

- `i = 2`
- `f = 3.1`

For getting other number types, the programmer needs to use the buit-in conversion function that will be described Section [...].

INI comes also with a `Char` type for single characters (letters and other ASCII characters) and a `String` type of character lists. Character literals are written between simple quotes and string literals are written between double quotes.

- `c = 'a'`
- `s = "hello"`

3.2 Functions

3.2.1 Syntax

Functions are defined with the `function` keyword. Functions have a name that must be unique. The syntax is the following:

```
<function> := function <name>(<parameters>) { <rules> }
```

Where `<name>` is the function name (an unique identifier), `<parameters>` is a coma-separated list of parameter identifiers, and `<rules>` is a list of rules that defines the function behavior. INI allows for rule-oriented programming. However, since rules are gathered into functions, it also allows for functional-like programming. The rule syntax is the following:

```
<rule> := <guard> { <body> }
```

Thus, the "expanded" function syntax is:

```
function <name>(<parameters>) {
  <guard1> { <body1> }
  <guard2> { <body2> }
  ...
  <guardN> { <bodyN> }
}
```

3.2.2 Parameters

Functions takes parameters which have a unique name within the function scope. Parameters are passed by reference and not by value. If the programmer wants to pass by value, it can be done using the `copy` built-in function. For instance, the function:

```
function f(a,b,c) { ... }
```

Can be invoked with `f(1,2,"abc")`, if `f` expects two integers and one string. Parameters can have default values. For instance, the `a` and `c` parameters may have default values:

```
function f(a,b=0,c="") { ... }
```

In that case, some parameter values are optional when invoked. For instance, `f(2,1)` invokes `f` with an empty string for `c`.

3.2.3 Returned values

A function can return a value by using a `return [<expr>]` statement within a rule body. If `<expr>` is not defined or if no return statements appear in the function, then the function returns a `Void` value. For instance:

```
1 function f(a,b=0,c="") {
2   ...
3   <guard> {
4     ...
5     return b
6   }
7   ...
8 }
```

The return statement at line 5 indicates that the function returns a value of type `Int` (since `b` is an integer, as seen in the parameter's default value at line 1). Note that return statements are optional when the function returns no values, but must always be the last statement of a rule.

3.2.4 Function types

A function type is noted as `(T1,T2,...TN)->T`, where `Ti` is the expected type of the ith parameter, and `T` is the function's return type.

3.2.5 Built-in Functions

Here is the list of built-in functions provided by INI. Built-in function have predefined function types, which are given here.

- `clear(v:T)->Void`: initializes or re-initializes a `v` variable, passed as a parameter.
- `copy(v:T)->T`: copies the content of a `v` variable, passed as a parameter.
- `eval(f:F,p1:T1,p:T2,...pN:TN)->T`: evaluates a function of type `F` passed as a reference, passing the parameters `pi` and returns the value returned by the evaluated function.
- `error(message:String)->Void`: throws an error with the given message. Throwing an error will stop the evaluation, unless it is caught by an error-event-triggered rule.
- `print(v:T)->Void`: writes the textual representation of the passed data on the standard output stream.
- `println(v:T)->Void`: writes the textual representation of the passed data on the standard output stream, and adds a carriage return at the end of the line.

- `read_keyboard()->String`: reads a line typed in by the user on the standard input stream and returns it.
- `size(Map(K,V))->Int`: returns the size of a map, i.e. the number of elements in it.
- `time()->Long`: returns the current system clock time as a `Long`.
- `to_byte(v:T)->Byte`: converts the given variable to a byte value.
- `to_double(v:T)->Double`: converts the given variable to a double value.
- `to_float(v:T)->Float`: converts the given variable to a float value.
- `to_int(v:T)->Int`: converts the given variable to a integer value.
- `to_long(v:T)->Long`: converts the given variable to a long value.
- `to_string(v:T)->String`: converts the given variable to a string value.

3.2.6 Bindings to Java objects

INI only provides a minimal set of built-in functions. For all other functions, one can bind new functions to Java APIs. Thus, to use Java objects from INI the programmers just need to define bindings from INI functions to Java constructors, methods or fields. The binding syntax is the following:

```
<name>(<types>) -> <type> => "string1", "string2"
```

This binding declares a new function named `<name>`, that takes parameters typed with the given coma-separated type list (`<types>`) and returns a typed result. The corresponding Java element that will be used when invoking the function is defined thanks to the two strings following the `=>` binding operator, where `string1` is the target Java class fully-qualified name, and where `string2` is one of the following:

- the target field name (belonging to the class),
- the target method name (belonging to the class) followed by "(..)" to indicate that it is a method (and not a field),
- "new(..)" to indicate that the target is a constructor of the class.

It is not needed to specify if the Java method or field is static, since INI will determine it automatically depending on the parameter types of the function. Non-static members will require to pass an instance of the type of the target class as the first parameter.

For instance, the following code defines two bound functions to call the classical `System.out.println(..)` method in Java. The `out()` function binds to the static `System.out` field, and the `java_println()` function binds to the `Writer.println(String)` non-static method.

```
out()->Writer => "java.lang.System", "out"
java_println(Writer,String)->Void => "java.io.Writer", "println(..)"
```

Programmers can then invoke both functions, which are well-typed thanks to the binding declarations.

```
java_println(out(),"hello Java")
```

3.3 Rules

3.3.1 Syntax

Each rule has the following syntax:

```
<rule> := <guard> { <body> }
<body> := <statements> <return_statement_opt>
<guard> := <event_matcher>
         | <logical_expr>
         | <event_matcher> <logical_expr>
```

The guard is either a event matcher (for event-triggered rules) or a logical expression, or both. Events will be depicted in the next section, and we will now focus on the logical expression part of the rule.

A logical expression within a guard is formed of variable accesses, function invocations, and literals, composed together with logical operators and/or comparison operators.

3.3.2 Logical operators

Allowed logical operators within a guard are:

- `<expr> && <expr>`: logical and, that applies to two boolean logical expressions. Like many languages, when the left-hand-side expression evaluated to false, the right-hand one is not evaluated.

- `<expr> || <expr>`: logical or, that applies to two boolean logical expressions, and that return true without evaluating the right expression if the left one evaluates to true.

- `! <expr>`: the not operator has two meanings depending on the type of the given expression. When `<expr>` is boolean, it is the logical inversion operator. In all other cases, the not operator will test the existence of a resulting value when evaluating the expression. To use a Java analogy, it would be similar to `<expr> == null`. Conversely, `<expr>` in INI will equal to `!!<expr>` or, in Java, to `<expr> != null`. Note that if `<expr>` is an access to an undefined variable, then the resulting value will be `null` as well.

To illustrate the use of the not operator on non-boolean expression, let us take the following function:

```
1  function ping_pong() {
2    !v {
3      println("ping")
4      v=""
5    }
6    v {
7      println("pong")
8      clear(v)
9    }
10 }
```

This function triggers an infinite loop (it is actually a ping-pong effect between the two rules). When entering `ping_pong`, the `v` variable is always undefined since it is not a parameter of the function. Thus, the guard `!v` at line 2 evaluates to `true`, while the guard `v` at line 6 evaluates to `false`. So, INI evaluates the first rule and sets `v` to an empty string. Since `v` is not undefined anymore, the first rule cannot apply and the second one applies, leading to clearing the content of `v` at line 8. Then, the first rule can apply again, starting the endless ping-pong game over again. Note that the rules order is not important. The following program gives exactly the same result:

```
1  function ping_pong() {
2    v {
3      println("pong")
4      clear(v)
5    }
```

```
6     !v {
7       println("ping")
8       v=""
9     }
10  }
```

3.3.3 Comparison operators

Comparison operators are binary operators that apply to objects. In INI, using a comparison operator between two object implies that they are of the same type. If not, a typing error will occur.

- `<expr> == <expr>`: compares two objets and returns true if equal in terms of values.
- `<expr> != <expr>`: compares two objets and returns false if equal in terms of values.
- `<expr> > <expr>`: used mainly to compare two numbers, but can actual apply to any comparable objets.
- `<expr> >= <expr>`: used mainly to compare two numbers, but can actual apply to any comparable objets.
- `<expr> < <expr>`: used mainly to compare two numbers, but can actual apply to any comparable objets.
- `<expr> <= <expr>`: used mainly to compare two numbers, but can actual apply to any comparable objets.

3.3.4 Regular expression match operator

In order to match strings in a concise way, INI provides a match operator $\sim$, with can match a string against a regular expression [2] and bind matching groups (if any) to INI variables. For example:

`"a b c"` $\sim$ `regexp("(.)  (b) (.)",v1,v2,v3)` will match and be evaluated to `true`. Since there are three groups (between parenthesis), the three match sub-results will be bound to the given variables `v1`, `v2`, and `v3`. Thus, once the match is done, we will have `v1="a"`, `v2="b"`, and `v3="c"`. For more information on regular expressions as used in INI, read the Javadoc for the `java.regexp.Pattern` class and refer to [2].

Example:

```
1  function greetings(sentence) {
2    sentence ~ regexp("Hello (.*)",name)
3      || sentence ~ regexp("Hi (.*)",name) {
4      println("Hello to "+name)
5      return
6    }
7    sentence ~ regexp("Bye (.*)",name)
8      || sentence ~ regexp("See you (.*)",name) {
9      println("Bye to "+name)
10     return
11   }
12 }
```

This function matches the given sentence to determine who it is said hello or bye to. Typically, the invocation `greetings("See you Renaud")` will print out `"Bye to Renaud"`. Note the use of the return statement at lines 5 and 10 to ensure that rules are applied once at best.

3.3.5 Event-triggered rules

Besides rules that are applied because the logical expression of the guard evaluates to true, some rules can also be event-triggered. An event rules takes configuration parameters as an annotation, and will provide input arguments when fired. Thus, an event-rule guard starts with one event expression, of the form `@<event>(<input arguments>) [<configuration parameters>]`. Configuration parameters may be optional depending on the event. Some events are evaluated synchronously (in the function/process thread), but some event are fired asynchronously and concurrently (multi-threaded evaluation).

3.4 Synchronous events

Synchronous events are evaluated within the function evaluation thread. When a synchronous event evaluates, no other rule or event can be evaluated concurrently. In functions, only synchronous events are allowed. To use asynchronous events, the programmers must define a process, as explained in section [...].

3.4.1 `@init` event

As the name says, `@init`-trigger rules are invoked when the function starts evaluating. It is the right place to initialize the variables that may need to be used within the other rules. This event takes no parameters.

For example, to repeat a rule n times:

```
function f(n) {
  @init() {
    i=0
  }
  i < n {
    // repeated code here
    ...
    i++
  }
}
```

Note that event-based rules can also have a predicate for triggering the rule or not. For instance, if `n <= 0`, one may want to stop the function evaluation and print out a message:

```
function f(n) {
  @init() && n<=0 {
    println("wrong repeat value")
    return
  }
  @init() && n>0 {
    i=0
  }
  i < n {
    // repeated code here
    ...
    i++
  }
}
```

3.4.2 `@end` event

On the contrary to the `@init` event, `@end` is triggered when no more rule apply, and when the function is about to return (not including explicit return statements, which do no trigger the `@end` event). Note that any state change

in an `@end`-triggered rule body will never lead to any other rule re-evaluation, even if the state change makes an existing rule applicable again.

3.4.3 `@update` event

The `@update(old_value, new_value) [variable = <variable>]` event occurs when the given variable is modified by the program (i.e. by one of the evaluated rules). For instance, `@update(b,c) [variable = a]` is triggered if the value of `a` is modified. The old value of `a` will be kept in the variable `b` and the new value of `a` will be kept in the variable `c`.

3.4.4 `@error` event

The `@error[message:String]()` event occurs when an error is thrown during the function's evaluation.

3.5 Processes

Processes are very similar to functions, except that they run asynchronously and concurrently.

3.5.1 Getting started with processes

Within a process, it is allowed to to use asynchronous events, which are also run concurrently to other events within the process. An example of an asynchronous event is the `@every` event, which is triggered at regular intervals defined by the `time` annotation parameter. For example, to define a process that says `"hello"` every second:

```
1 process p() {
2  @every() [time=1000] {
3   println("hello")
4  }
5 }
```

Processes are spawned just by invoking them as regular functions. The main difference is that this invocation is not blocking by default. For instance, to start the previously defined process, just write:

```
1 process main() {
2  @init() {
3   p()
4  }
```

```
}
```

Processes may take arguments exactly like functions. The following program will write `"hi"` every second:

```
process main() {
 @init() {
  p("hi")
 }
}
process p(msg) {
 @every() [time=1000] {
  println(msg)
 }
}
```

3.5.2 Inter-process communication

Process communicate with each other through channels. A channel has a unique name and can be used to produce and consume data. To produce data in a channel, a function or a process must invoke the `produce` built-in function. This is a non-blocking call. On the other end of the channel, a process must consume the data using the `@consume` event. Since it is an asynchronous event, only processes can use the `@consume` event. For instance, the following program starts a process `p` and send a `"hello"` message through a channel `c`. Note that the `p` process never terminates and will wait on subsequent data on the channel `c` to be produced.

```
process main() {
 @init() {
  p()
  produce("c", "hello")
 }
}
process p() {
 @consume(msg) [channel="c"] {
  println(msg)
 }
}
```

3.5.3 Stopping processes

Since processes usually never end, it may be useful to force them to terminate. This can be achieved by terminating all the events in the process, which can be done with the `stop` built-in function. This function will take a parameter, which is an event rule referred to thanks to the name of the rule. For instance, to name a `@consume` event rule `c`, just write: `c: @consume(...)`.

Once all the event rules are terminated, the process terminates and the `@end()` event will be triggered. As an example, the following program launches a process that will print an `"hello"` message an will terminate right after:

```
1  process main() {
2   @init() {
3    p()
4    produce("c", "hello")
5   }
6  }
7  process p() {
8   c: @consume(msg) [channel="c"] {
9    println(msg)
10   stop(c)
11  }
12  @end() {
13    println("bye")
14  }
15 }
```

3.6 Asynchronous events

Asynchronous events can only be used in process definitions, and are evaluated concurrently with other rules or events in the process.

3.6.1 `@every` event

The `@every() [time = interval:Int]` event occurs every `interval` milliseconds.

3.6.2 `@cron` event

The `@cron() [pattern:String]` event occurs on times indicated by the `pattern` UNIX CRON pattern expression. CRON is a task scheduler that allows the concise definition of repetitive task within a single (and simple) CRON pattern [1]. A UNIX crontab-like pattern is a string split in five space separated parts. Each part is intended as:

1. Minutes sub-pattern. During which minutes of the hour the event should occur? The values range is from 0 to 59.
2. Hours sub-pattern. During which hours of the day should the event occur? The values range is from 0 to 23.
3. Days of month sub-pattern. During which days of the month should the event occur? The values range is from 1 to 31. The special value "L" can be used to recognize the last day of month.
4. Months sub-pattern. During which months of the year should the event occur? The values range is from 1 (January) to 12 (December), otherwise this sub-pattern allows the aliases "jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", "nov" and "dec".
5. Days of week sub-pattern. During which days of the week should the event occur? The values range is from 0 (Sunday) to 6 (Saturday), otherwise this sub-pattern allows the aliases "sun", "mon", "tue", "wed", "thu", "fri" and "sat".

Some examples:

- `"5 * * * *"`: This pattern causes the event to occur once every hour, at the begin of the fifth minute (00:05, 01:05, 02:05 etc.).
- `"* * * * *"`: This pattern causes the event to occur every minute.
- `"* 12 * * Mon"`: This pattern causes the event to occur every minute during the 12th hour of Monday.
- `"* 12 16 * Mon"`: This pattern causes the event to occur every minute during the 12th hour of Monday, 16th, but only if the day is the 16th of the month.
- `"59 11 * * 1,2,3,4,5"`: This pattern causes the event to occur at 11:59AM on Monday, Tuesday, Wednesday, Thursday and Friday.
- `"59 11 * * 1-5"`: This pattern is equivalent to the previous one.

3.7 Maps, lists, and sets

3.7.1 Map definition and access

Maps are natively supported by INI. A map is automatically defined when accessed through the map access expression that uses square brackets: `map[key]`. Keys and values within maps are of any type, but shall all be of the same type for a given map. For instance the following statement list is valid:

```
m["key1"] = 1
m["key2"] = 2
println(m["key2"])
```

On the other hand, the following statement list is not valid because the first line initializes `m` to be a map of integer values accessed through string keys and:

- the key is an integer at line 2,
- the value is a string at line 3,
- there are no typing errors at line 4.

```
1 m["key1"] = 1
2 m[1] = 2
3 m["key2"] = "test"
4 println(m["key2"])
```

Note that the size of a map (i.e. the number of elements in it) can be retrieved with the `size` built-in function (see Section [...]). Emptying a map or removing an entry is done with the `clear` function (not implemented yet).

3.7.2 List definition and access

In INI, a list is simply a map where keys are integers.

3.7.3 Integer sets

To allow easy iteration on lists, INI provides a constructor for set of integers. A set of integers is defined with two bounds: `[min..max]`. For instance, the set that contains all the integers between `0` and `10` (bounds included) is written `[0..10]`.

3.7.4 Set selection expressions

Set can be used in set selection expressions that allows the programmer to randomly select elements in a set and bind their values to local variables. A selection condition must be used to select the elements upon a given criteria. For instance, to select two integers `i` and `j` that are contained within 0 and 10, so that `i ¡ j`, one can write the following set selection expression:

```
i, j of [0..10] | i < j
```

Set selection expressions must be used in guards. The difference between a set selection guard and a regular guard is that the former one will be evaluated to true until all the possible values have been picked from the set. In other words, the guard will stop matching only once none of the set elements fit the selection condition.

Example: sorting the elements of a list (see Section [...]).

```
1 function sort1(s) {
2   i of [0..size(s)-2] | s[i] > s[i+1] {
3     swap(s[i],s[i+1])
4   }
5   @end() {
6     return s
7   }
8 }
```

As we will see in the next section, set selection expressions can be used to select instances of types that have been constructed by the program. This feature requires the use of user-defined types.

3.8 User-defined types

A user-defined type is a way for the programmer to define complex structures. Structures contains values stored in named fields, which need to be explicitly typed when defined by the programmer. Section [...] gives an example that uses types.

3.8.1 Simple structured types

To define a new type, the programmer uses the `type` keyword and sets a name that must be unique, and a set of typed field. Note that in INI, type name must start with an uppercased letter. For example, to define a `Person` type that has a name and an age, one may want to write:

```
type Person = [name:String, age:Int]
```

To use a type, the programmer must instantiate the type using one of its constructor. For a simple type, there is only one constructor named by the name of the type. So, for constructing a new person and store it in a `p` variable:

```
p = Person[name="Renaud", age=37]
```

Later on, the program can access the object's fields using the dotted notation:

```
println(""+p.name+" is "+p.age)
```

Note that field initialization is not mandatory. For instance, one can construct a person with undefined age or name.

Finally, types can be recursively defined. For instance, one can use the `Person` type within the `Person` type definition itself. For instance, we may want to add two fields for the parents of the person:

```
type Person = [name:String, age:Int,
               father:Person, mother:Person]
```

We may even want to add the children, as a person list:

```
type Person = [name:String, age:Int,
               father:Person, mother:Person,
               children:Person*]
```

Note that when using such types, INI checks that the used objects work on the right fields with the correct types.

3.8.2 Algebraic types

Algebraic types are an extension of simple structured type that allow the definition of types that have different constructors. Algebraic types in INI are not actual pure algebraic types since they contain named fields that do not need to be initialized. However, they are defined and used in a very similar way, as shown in the Fibonacci example of Section [...]. Algebraic types can be related to AST (Abstract Syntax Trees) in the sense that they allow the definition of typed trees structures.

For instance, we can define a type for expressions that include typical arithmetic operations on floating point numbers.

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

```
            | Mult[left:Expr, right:Expr]
            | Div[left:Expr, right:Expr]
            | Minus[left:Expr, right:Expr]
            | UMinus[operand:Expr]
```

Once define, the programmer can instantiate any expression using the constructors. For example to instantiate the expression `-(3.0*2.0+1.0)`:

```
1 expr = UMinus[operand=Plus[
2   left=Mult[
3     left=Number[value=3.0],
4     right=Number[value=2.0]],
5   right=Number[value=1.0]]]
```

3.8.3 Type set selection expressions

In Section 3.7.4, we have seen the set selection expression that allows to select values within a set. Each object constructed with a user type constructor is automatically part of an instance set, which is named after the constructor name. Thus, it is possible to select instances using the set selection construct. For example, the following rule raises an error if a number has a undefined value:

```
n of Number | !n.value {
    error("invalid number value")
}
```

Note that sets are local to the functions that construct the instances.

3.8.4 Type set match expressions

In Section 3.3.4, we have seen the match operator (∼) for string, which is based on regular expressions. The match operator can also be used to match type instances. It an be used similarly to the match construction in the CAML language. For example, we can use the match operator to implement a basic expression printer (see function `expr_str` at line 19) and evaluator (function `expr_val` at line 40).

```
1 type Expr = Number[value:Float]
2           | Plus[left:Expr, right:Expr]
3           | Mult[left:Expr, right:Expr]
4           | Div[left:Expr, right:Expr]
5           | Minus[left:Expr, right:Expr]
6           | UMinus[operand:Expr]
```

```
7
8  function main() {
9    @init() {
10     expr = UMinus[operand=Plus[
11       left=Mult[
12         left=Number[value=3.0],
13         right=Number[value=2.0]],
14         right=Number[value=1.0]]]
15     println("The value of "+expr_str(expr)+" is "+expr_val(expr))
16   }
17 }
18
19 function expr_str(expr) {
20   expr ~ Number[value] {
21     return to_string(expr.value)
22   }
23   expr ~ Plus[left,right] {
24     return "("+expr_str(expr.left)+"+"+expr_str(expr.right)+")"
25   }
26   expr ~ Mult[left,right] {
27     return "("+expr_str(expr.left)+"*"+expr_str(expr.right)+")"
28   }
29   expr ~ Minus[left,right] {
30     return "("+expr_str(expr.left)+"-"+expr_str(expr.right)+")"
31   }
32   expr ~ Div[left,right] {
33     return "("+expr_str(expr.left)+"/"+expr_str(expr.right)+")"
34   }
35   expr ~ UMinus[operand] {
36     return "-("+expr_str(expr.operand)+")"
37   }
38 }
39
40 function expr_val(expr) {
41   expr ~ Number[value] {
42     return expr.value
43   }
44   expr ~ Plus[left,right] {
45     return expr_val(expr.left)+expr_val(expr.right)
46   }
47   expr ~ Mult[left,right] {
48     return expr_val(expr.left)*expr_val(expr.right)
49   }
50   expr ~ Minus[left,right] {
51     return expr_val(expr.left)-expr_val(expr.right)
52   }
53   expr ~ Div[left,right] {
54     return expr_val(expr.left)/expr_val(expr.right)
55   }
```

```
56    expr ~ UMinus[operand] {
57      return -expr_val(expr.operand)
58    }
59  }
```

3.9 Function references and `eval`

Functions can be referenced and passed as parameters to other functions. A function reference expression is defined with the function keyword. For instance: `f = function(my_function)` creates a reference to `my_function` and store it in the `f` variable. Then the referenced function can be evaluated using the built-in function `eval`, that takes a reference to the function to be evaluated followed by the parameters to be passed to the evaluated function.

The following program illustrates the use of function references. It defines a `sort` function at line 9 that takes a list `l` to sort and a comparison function to compare the elements of the list. This function is passes in the `comparator` parameter.

```
1  function main() {
2    @init() {
3      l = "a string to sort"
4      sort(l,function(compare))
5      println("result: "+l)
6    }
7  }
8
9  function sort(l, comparator) {
10   i of [0..size(l)-2] | eval(comparator,l[i],l[i+1]) > 0 {
11     swap(l[i],l[i+1])
12   }
13 }
14
15 function compare(e1,e2) {
16   e1 > e2 { return 1 }
17   e1 < e2 { return -1 }
18   e1 == e2 { return 0 }
19 }
```

As one can see at line 10, that we invoke the `eval` function to call the function referenced in `comparator`. This function takes two elements (here `l[i]` and `l[i+1]` are passed) and returns an integer (since the result of `eval` is compared with 0). At line 4, we can see that the passed comparison function is the `compare` function defined at line 15, however, any comparison

function that takes to elements to compare and returns an integer may be used as a comparator.

3.10 Imports

Since all the functions should not be defined within a single file, INI programs can start with a list of import clauses. For example:

```
import "ini/examples/lib_io.ini"
```

Imports allow the definition and use of function libraries. In particular, it is recommended to define bindings (see Section **??**) within external files and to import them when required.

3.11 User-defined events

Besides the built-in events, INI allows developers to create their own events. For example, we will try to create a simple custom event. This event will copy one file to another file. We need two in parameters: one for source file name and another for destination file name. Besides, we will store the time needed for copy process in the variable `v` and the result of this process (whether successful or not) in the variable `s`. In INI, `t` and `s` are examples of out parameters.

```
1
2  package ini.ext.events;
3
4  import ini.ast.AtPredicate;
5  import ini.ast.Rule;
6  import ini.eval.IniEval;
7  import ini.eval.at.At;
8  import ini.eval.data.Data;
9  import ini.eval.data.RawData;
10
11 import java.io.File;
12 import java.io.FileInputStream;
13 import java.io.FileOutputStream;
14 import java.io.InputStream;
15 import java.io.OutputStream;
16 import java.util.HashMap;
17 import java.util.Map;
18
```

```
19 public class AtCopyFile extends At {
20   public String sourceFileName;
21   public String destFileName;
22   public long copyTime;
23   public boolean success = false;
24   Map<String, Data> variables = new
25         HashMap<String, Data>();
26   Map<Thread, At> threadAt = new HashMap<Thread, At>();
27   @Override
28   public void eval(final IniEval eval) {
29     final Data sfn = getInContext().get("source");
30     sourceFileName = (String) sfn.getValue();
31     final Data dfn = getInContext().get("destination");
32     destFileName = (String) dfn.getValue();
33     new Thread() {
34       @Override
35       public void run() {
36         try {
37           long beginTime = System.currentTimeMillis();
38           File sourceFile = new File(sourceFileName);
39           File destFile = new File(destFileName);
40           InputStream in = new
41                       FileInputStream(sourceFile);
42           OutputStream out =
43                       new FileOutputStream(destFile);
44           byte[] buffer = new byte[1024];
45           int fLength;
46           while ((fLength = in.read(buffer)) > 0) {
47             out.write(buffer, 0, fLength);
48           }
49           in.close();
50           out.close();
51           long endTime = System.currentTimeMillis();
52           copyTime = endTime - beginTime;
53           success = true;
54           variables.put(getAtPredicate().outParameters.
55                       get(0).toString(),
56               new RawData(copyTime));
57           variables.put(getAtPredicate().outParameters.
58                       get(1).toString(),
```

```
59                  new RawData(success));
60              this.setName("Demo copy thread");
61              execute(eval, variables);
62              terminate();
63            } catch (Exception e) {
64              e.printStackTrace();
65            }
66          }
67        }.start();
68      }
69  }
```

Now we can write a simple INI program to test this event.

```
1  @copy_file[String, String](Long, Boolean)=>
2                                  "ini.ext.events.AtCopyFile"
3  function main() {
4   @copy_file(t,s)[source = "file.exe",destination =
5                   "fileCopy.exe"] {
6    println("Time for copy: " + t + " milliseconds")
7   }
8  }
```

Before using the event, we need to declare it as you see in the first line of our program. The types of configuration and out parameters should be declared. In our example, we have two in parameters with type `String`. Besides, we have two out parameters, one with type `Long` and another with type `Boolean`. Moreover, a source code associated with the event also must be declared. When we run this program, it will copy a file `file.exe` to a file `fileCopy.exe`. The time needed for this process (in milliseconds) also will be displayed.

4 The INI type system

4.1 Definitions

INI comes with 5 built-in types for numbers: *Double*, *Float*, *Long*, *Int*, and *Byte*. They differ on the number of bytes used to encode them (same as Java encoding). INI comes also with a *Char* type for single characters (letters and other ASCII characters) and a *Boolean* type that takes only *true* and $false$ values. Finally, INI provides built-in map types: $Map(K, V)$. Map types are dependent types with a key type K and a value type V. When

$K = Int$, it means that the type is a list in the INI definition (in reality, it is more of an indexed set). Syntactically, lists can be noted with the * notation:

$$T* \hat{=} Map(Int, T)$$

A list of $Char$ (i.e. a $Char*$) is a $String$ type (note that $String$ and $T*$ types are actually aliases and do not exist as real types in the INI internal type representation).

$$String \hat{=} Char* \hat{=} Map(Int, Char)$$

INI types can be ordered with a type relation $\succ$. $A \succ B$ means that A is a super-type of B, i.e. A is assignable from B, whereas the contrary is not true. By default, number types in INI are ordered so that it is not possible to assign more generic numbers to less generic numbers.

$$Double \succ Float \succ Long \succ Int \succ Byte$$

4.2 Type Inference

As said in Section 5.1, 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 [3]. The typing algorithm is enhanced with polymorphic function support, abstract data types (or algebraic types) support, and with internal sub-typing 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.

4.3 Type Inference rules

4.3.1 Initiation rules

In order to infer types for the program, we first need some initiation rules. Here is a rule that states that undefined values have any type, i.e. their type is an unconstrained type variable.

$$\text{(NULL)}\ \frac{}{\vdash null : T}$$

Note that INI also has an internal `Void` type for functions returning no values. We then have some rules for literals, which can directly be inferred as resolved types:

- string literals such as `"abc"` will be typed as `String`,
- character literals such as `'a'` will be typed as `Char`,
- float literals such as `3.14` will be typed as `Float`,
- integers such as `1` will be typed as `Int`,
- `true` and `false` literals will be typed as `Boolean`,

4.3.2 Assignments

When an assignment is made from a resolved type (for instance from a literal, such as `x=2.0`), then the type of the assignee equals to the type of the assignment (for instance `x:Float`).

$$\text{(ASSIGNDEF)}\ \frac{\vdash x : T \qquad \vdash y : T}{\vdash x = (y : T)}$$

When the assignment is made from an unresolved type (e.g. `x=y`), then the type of the assignee is a super-type for the type of the assignment.

$$\text{(ASSIGN)}\ \frac{\vdash x : U \qquad \vdash y : V}{\vdash x = y}\ [U \succeq V]$$

4.3.3 Logical operators

Logical operators are easy to deal with since the result must be of the `Boolean` type. There are no constraints on the operands since undefined values mean `false` and defined ones mean `true` from a boolean perspective.

$$\text{(LCOMP)}\ \frac{\vdash x : T \qquad \vdash y : U}{\vdash (x\ \texttt{op}\ y) : Boolean}$$

$$\text{with } \texttt{op} \in \{\texttt{\&\&}, \texttt{||}\}$$

4.3.4 Comparison operators

Comparators imply that the result must be `Boolean`. Operands must be comparable, which is the case for all objects in INI. Moreover, we enforce that the compared object must be of the same type.

$$\text{(COMP)}\ \frac{\vdash x : T \qquad \vdash y : T}{\vdash (x\ \texttt{op}\ y) : \mathit{Boolean}}$$

$$\text{with } \texttt{op} \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}$$

4.3.5 Division

Division must occur between two expressions of the same type, which both must be subtypes of `Double` (e.g. `Float` or `Int`). Division returns a `Float` whatever the operands' type is, thus avoiding loss of precision for integers.

$$\text{(DIV)}\ \frac{\vdash x : T \qquad \vdash y : T}{\vdash (x\ \texttt{/}\ y) : \mathit{Float}}\ [T \preceq \mathit{Double}]$$

4.3.6 Multiplication and minus

Multiplication and minus operations must occur between two expressions of the same type, which both must be subtypes of `Double` (e.g. `Float` or `Int`). They will return the same type as the operands' type.

$$\text{(MULTMINUS)}\ \frac{\vdash x : T \qquad \vdash y : T}{\vdash (x\ \texttt{op}\ y) : T}\ [T \preceq \mathit{Double}]$$

$$\text{with } \texttt{op} \in \{\texttt{-}, \texttt{*}\}$$

4.3.7 The plus operator

There are two case for the plus operator. In the first case, it is uses as a string concatenation operator. For that, like in Java, the first operand must be of a resolved `String` type. If it is the case, the type of the second operand does not matter and the overall expression will be typed as a `String` and the `String` type will propagate. Typically, the `"a"+b` expression is a string concatenation case.

$$\text{(PLUS1)}\ \frac{\vdash x : \mathit{String} \qquad \vdash y : T}{\vdash (x\ \texttt{+}\ y) : \mathit{String}}$$

The second case occurs when the type of the first operand's type is undetermined. We then fall back to the arithmetic plus, which behaves exactly like $MULTMINUS$ in terms of typing.

$$\text{(PLUS2)}\ \frac{\vdash x : T \qquad \vdash y : T}{\vdash (x\ \texttt{+}\ y) : T}\ [T \preceq \mathit{Double}]$$

4.3.8 Unary minus (sign inversion)

The unary minus operator expects an operand which is a number and will return a result of the same type.

$$\text{(UMINUS)}\ \frac{\vdash x : T}{\vdash (\texttt{-}x) : T}\ [T \preceq \mathit{Double}]$$

4.3.9 Post operators

The post increment/decrement operators both expect an operand which is a number and will return a result of the same type.

$$\text{(POSTOP)}\ \frac{\vdash x : T}{\vdash (x\ \texttt{postop}) : T}\ [T \preceq \mathit{Double}]$$

$$\text{with } \texttt{postop} \in \{\texttt{++}, \texttt{--}\}$$

4.3.10 The not operator

The logical negation operator returns a boolean, but does not force any constraint on the operand (undefined means `false` and defined means `true` for non-boolean objects).

$$\text{(NOT)}\ \frac{\vdash e : T}{\vdash (\texttt{!}e) : \mathit{Boolean}}$$

4.3.11 The match operator

Within a guard a match operator can be used to match strings with regular expressions and bind the results to some variables. For instance `"a b c"` $\sim$ `regexp("(.) (b) (.)",v1,v2,v3)` will match and be evaluated to `true`. Since there are three groups (between parenthesis), the three match sub-results will be bound to the given variables. Thus, once the match is done, we will have `v1="a"`, `v2="b"`, and `v3="c"`. With regard to typing, all the bound variables are strings.

$$\text{(MATCH1)}\ \frac{\vdash x : String \qquad \vdash y : String \qquad \{\vdash v_i : String\}_{i\in[1..n]}}{\vdash (x \sim \texttt{regexp}(y, v_1, v_2, ..., v_n)) : Boolean}$$

A match operator can also be used to match a constructor's instance with some constraints given by boolean logical expressions on the constructor's fields.

$$\text{(MATCH2)}\ \frac{\{\vdash e_i : Boolean\}_{i\in[1..n]}}{\vdash (e \sim C[e_1, e_2, ..., e_n]) : Boolean}\ [\{freevars(e_i) = \emptyset\}_{i\in[1..n]}]$$

4.3.12 Map access

A map access node such as `x[y]` allows to say that the resulting type of the expression is `V`, if `x` is of type `Map(K,V)`.

$$\text{(MAPACCESS)}\ \frac{\vdash x : Map(K, V) \qquad \vdash y : K}{\vdash x[y] : V}$$

4.3.13 Field access

For a field access such as `x.f`, the type of the accessed expression `x` must have a field of the right name and type. Note that, when using match expressions, by construction, it is often the case that the type of the accessed expression is already resolved.

$$\text{(FIELDACCESS)}\ \frac{\vdash T.f : V}{\vdash (x : T).f : V}$$

4.3.14 Function invocation

For an invocation, the rule implies some constraints between the function type and the types of the result and of the parameter expressions. To support polymorphism, the function body is evaluated within its own environment, so that each invocation has its own set of constraints.

$$\text{(INVOCATION)}\ \frac{\vdash f : T_1, T_2, ..., T_n \to T \qquad \{\vdash e_i : T_i\}_{i\in[1..n]}}{\vdash f(e_1, e_2, ..., e_n) : T_1, T_2, ..., T_n \to T}\ [new(env)]$$

4.3.15 List definition

Lists definitions are done with the following syntax: `[e1, e2, ... en]`, where all the expressions must be of the same type `T`. Then the type of the resulting list is `T*` (equivalent to `Map(Int,T)`)

$$\text{(LISTDEF)}\ \frac{\{\vdash e_i : T\}_{i\in[1..n]}}{\vdash \texttt{[}e_1, e_2, ..., e_n\texttt{]} : T*}$$

4.3.16 Return statement

A return statement implies that the enclosing function's return type is `Void`. Note that, in addition, the function's return type is `Void` if no return statements appear in the function body.

$$\text{(RETURN)}\ \frac{\vdash f : \_ \rightarrow Void}{\vdash (\texttt{return}) \in f}$$

A return statement with an expression implies that the enclosing function's return type is of the expression's type.

$$\text{(RETURNEXPR)}\ \frac{\vdash f : \_ \rightarrow T \qquad \vdash e : T}{\vdash (\texttt{return}\ \ e) \in f}$$

4.3.17 Type instantiation

One can construct an instance of an algebraic type using one of its constructor. For instance, a algebraic type defined as: type `A = C[f1:T1, f2:T2, ... fn,Tn]` has one constructor `C` with n typed fields. A constructor `C` of an algebraic type `A` is typed with a type of the same name (`C`), with $C \preceq A$. When an instance of `A` is constructed using `C` through the expression `C[f1=e1,f2=e2, ... fn=en]`, then the type of each expression `ei` must be of the type of the field `fi`, as declared in the algebraic type definition (i.e. `Ti`).

$$\text{(CONSTRUCTOR)}\ \frac{\{\vdash C.f_i : T_i\}_{i\in[1..n]} \qquad \{\vdash e_i : T_i\}_{i\in[1..n]}}{\vdash \texttt{C[}f_1\texttt{=}e_1\texttt{,}f_2\texttt{=}e_2\texttt{,...}f_n\texttt{=}e_n\texttt{]} : C}$$

with T, the type that corresponds to the C constructor, which is part of an algebraic type

4.3.18 Integer set declaration

An integer set declaration allows the programmer to declare a integer domain with min and max bounds. The syntax is `[e1..e2]` where `e1` is the min bound and `e2` is the max one. Both min and max bound expressions must be of type `Int`, and the resulting type is an `Set(Int)`.

$$\text{(INTSET)}\ \frac{\vdash e_1 : Int \qquad \vdash e_2 : Int}{\vdash [e_1\,..\,e_2] : Set(Int)}$$

4.3.19 Set selection

Within a set expression, one can select elements in a set, and bind them to variables. In that construct, each variable is of the type of the set elements.

$$\text{(SELECTION1)}\ \frac{\{\vdash e_i : T\}_{i\in[1..n]} \qquad \vdash s : Set(T)}{\vdash e_1, e_2, ...e_n\ \texttt{of}\ s}$$

A selection operation can also be done within a constructor type `C`, i.e. it will select instances that has been constructed through this constructor.

$$\text{(SELECTION2)}\ \frac{\{\vdash e_i : C\}_{i\in[1..n]}}{\vdash e_1, e_2, ...e_n\ \texttt{of}\ (\texttt{C} : Set(C))}$$

5 Some INI examples

5.1 Counting occurences in a list

In order to illustrate the use of maps and lists with INI, we now show how to implement a simple function that counts the occurences of the elements within a given list. In order to access the counted results for each element, we store them in a map, where each entry's key is the counted element, and the entry's value is the calculated number of occurences of the element.

For implementing this function, we will use INI built-in map types, which allow the programmer to associate keys with values by using a square bracket based syntax (like arrays in most languages). Within an INI map, all the keys and values must be of the same type in order to avoid typing errors. In INI, a list is a `Map(Int,T)`, which can also be said as `T*` (list of `T`). A type alias is defined for strings, which are lists of characters: `String` $\widehat{=}$ `Map(Int,Char)`.

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.

```
1  function countOccurencesAndStoreToMap(l) {
2    @init() {
3      i=0
4    }
5    i < size(l) {
6      c[l[i++]]++
7    }
8    @end() {
9      return c
10   }
11 }
```

The `countOccurencesAndStoreToMap` function shown here simply iterates through the given `l` variable. As expected, this variable is inferred by INI to be a list, since it is used as such in the expressions `l[i++]` (line 6) and `size(l)` (line 5). So, for each element in `l`, the function accesses the corresponding entry in a `c` map: c[l[i]] and increments the value of this entry with the `++` operator (`c[l[i]]++`). Then, the program goes to the next element of the list with the `++` operator on `i` (hence the weird yet concise statement `c[l[i++]]++` of line 6). The `++` operator is a typical post increment operator as it is found in C or Java languages.

As shown in the following program, we can now use this function to count the occurences within a list of integers. Note that a program shall define a main function that is run by the INI interpreter.

```
1  function main() {
2    @init() {
3      l = [1, 2, 1, 7]
4      println("Counting '"+l+"'")
5      c = countOccurencesAndStoreToMap(l)
6      println("Number of 1: "+c[1])
7      println("Number of 7: "+c[7])
8      println("Number of 3: "+c[3])
9    }
10 }
```

Output:

```
Counting '[1,2,1,7](0..3)'
Number of 1: 2
Number of 7: 1
Number of 3: null
```

When executing the main function, the `c` variable is filled with the count of each element in the list `l`. Note that in that case, the type of `c` is `Map(Int,Int)`, because `l` is of type `Int*`. However, we can note that nothing prevents the program to count other types of elements. For instance, we can count a list of strings: `l = ["a","ab","abc","ab","a"]`. In that case, the type of `c` would be `Map(String,Int)`. Finally, since strings are defined as lists of characters in INI, we can also count letter occurences in a string, as showed below.

```
1  function main() {
2    @init() {
3      s = "This is the string we will count"
4      println("Counting '"+s+"'")
5      c = countOccurencesAndStoreToMap(s)
6      println("Number of e(s): "+c['e'])
7      println("Number of a(s): "+c['a'])
8      println("Number of s(s): "+c['s'])
9      println("Number of i(s): "+c['i'])
10     println("Number of spaces: "+c[' '])
11   }
12 }
```

Output:

```
Counting 'This is the string we will count'
Number of e(s): 2
Number of a(s): null
Number of s(s): 3
Number of i(s): 4
Number of spaces: 6
```

In terms of typing, it means that the `countOccurencesAndStoreToMap` function is *polymorphic*. A polymorphic function is more generic since it can be applied to several types. More precisely, in our case, the inferred functional type is `(Map('E,Int))->Void`, where `'E` is a type parameter. Actually, the INI type inference algorithm finds the most general type so that functions are only typed with the least possible constraints and remain polymorphic when possible. For more details, see [...].

5.2 Implementing a sort function

With INI, there is a very straightforward way to implement a sort function, which consists of taking advantage of *set expressions*. A set expression allows

to select arbitrary objects within a set. These selected objects may follow some criteria given in the second part of the expression. Set expressions are used in guard so that the guard is evaluated to true only if an object that matches the given criteria can be found in the set. Here, to implement the `sort1` function, we simply select all the indexes `i` in the `s` list so that `s[i]` is greater than `s[i+1]` (line 2). When this rule matches it simply swaps `s[i]` and `s[i+1]`, so that the list will eventually be sorted when the rule cannot apply anymore. In that case, the `@end` event is triggered and the function returns the sorted list.

```
1 function sort1(s) {
2   i of [0..size(s)-2] | s[i] > s[i+1] {
3     swap(s[i],s[i+1])
4   }
5   @end() {
6     return s
7   }
8 }
```

Note that this function is polymorphic. Its type is `(Map(Int,'T))->Map(Int,'T)`, or simpler written `('T*)->'T*`.

Of course, programmers can also use explicit indexes to iterate on the list to be sorted, leading to more classical code. The following function implements a basic bubble sort with INI. The iteration is done with two rules: one which swaps the current elements (line 7), and one which does not (line 12). A `swap` boolean flag is used to know if there was a swap during the iteration. If `swap` is true once the end of the list is reach, we set back the index `i` to `0` so that the iteration rules apply all over again (line 15).

```
1 function sort2(s) {
2   @init() {
3     i = 0
4     swap = false
5     size = size(s)-1
6   }
7   i < size && s[i] > s[i+1] {
8     swap(s[i],s[i+1])
9     swap = true
10    i++
11  }
12  i < size && s[i] <= s[i+1] {
13    i++
```

```
14    }
15    i==size && swap {
16      swap = false
17      i = 0
18    }
19    @end() {
20      return s
21    }
22  }
```

As we can see, the first implementation is more intuitive and more concise. The advantage with the second implementation is that the programmer can actually control the way the list is iterated on. With the set expression, there is no warranty on the order the elements will match, which could lead to performance issues. Here the two implementations have a similar complexity of $0(n^2)$. In order to get better complexity, we can for instance implement the well-known Quicksort algorithm [...], as shown here.

```
1   function quicksort(s,lo,hi){
2     hi>lo && !done{
3       p = partition(s,lo,hi,lo)
4       quicksort(s, lo, p-1)
5       quicksort(s, p+1, hi)
6       done = true
7     }
8     @end() {
9       return s
10    }
11  }
12
13  function partition(s,lo,hi,pivotIndex){
14    @init() {
15      pivotValue = s[pivotIndex]
16      swap(s[pivotIndex],s[hi])
17      index=lo
18      i=lo
19    }
20    i<hi && s[i]<=pivotValue {
21      swap(s[i],s[index])
22      index++
23      i++
```

```
24    }
25    i<hi && s[i]>pivotValue {
26      i++
27    }
28    @end() {
29      swap(s[hi], s[index])
30      return index
31    }
32  }
```

This algorithm is more complex to understand but it performs very well for middle-sized lists. The interesting characteristic to point out here is that the quicksort function is recursive. Hence, INI allows both rule-based programming style and functional programming style. Programmers may use one or the other depending on the type of problem they need to solve.

5.3 Constructing a Fibonacci tree

A Fibonacci tree is a binary tree defined as follows:

- each node contains an integer value v
- if $v > 2$, the values of the two sub-nodes are $v - 1$ and $v - 2$
- if $v = 2$, the values of the two sub-nodes are 1

Fibonacci trees have interesting properties to calculate the Fibonacci series. In particular, for a given node having the value v, the corresponding Fibonacci number equals to the number of tree leaves under this node.

In order to construct a Fibonacci tree, the best way is to define a tree type. In INI, it is possible to define structural types that contain fields. Typically, we can define a `Node` type as follows:

```
type Node = [value:Int, left:Node, right:Node]
```

As shown in the code, a node has three fields: an integer value and two nodes corresponding to the left and right sub-nodes (the `Node` type is recursively defined). Once the programmer has defined such a type, it is possible to construct new instances of it with the following syntax: for instance, `node = Node[value=1]`. This statement creates a new `Node` object and initializes the value field to 1. In INI, it is not mandatory to define all the fields when instantiating a type, it simply means that the not initialized fields are undefined (`null` value). To access the field value of a given type instance, the programmer must use the doted notation. For instance: `i = node.value`

Before starting the implementation, it is important to note that when the `Node` type is instantiated, the resulting object is part of a `Node` set since INI ensures a correspondence between the constructors and the set of objects that has been constructed through them. This can be useful to write guard with set expressions upon type instances.

```
1  function fibtree1(v) {
2    @init() {
3      root = Node[value=v]
4    }
5    n of Node | n.value > 2 && !n.left {
6      n.left = Node[value=n.value - 1]
7      n.right = Node[value=n.value - 2]
8    }
9    n of Node | n.value == 2 && !n.left {
10     n.left = Node[value=1]
11     n.right = Node[value=1]
12   }
13   @end() {
14     return root
15   }
16 }
```

The `fibtree1` function proposes a first implementation that uses the `Node` type and two set expression to construct a tree from a given root value. It takes the initial value `v` as an argument and first creates the root node of the tree (line 3) with undefined sub-nodes. Then, it defines two rules: one for constructing the sub-nodes for all the nodes having a value greater than 2 (line 5), one for constructing the final nodes when the value equals 2 (line 9). Note the use of `!n.left` in the two guards. It ensures that the rule will never apply again to a node that has already been handled (i.e. left sub-nodes is already there). Actually, the complete guard should be `!n.left && !n.right` but it is not necessary since the program ensures that the left and right sub-nodes and always constructed at the same time. Finally when no more rule applies, i.e. all the possible nodes have been constructed, the function returns the root node of the tree.

This implementation works fine, but from a typing perspective, it would be nicer to have different constructors for the regular nodes and for the leaf nodes, since the latter can never have sub-nodes. It would allow better typing of the objects. This can be achieved using algebraic data types, which are supported by INI. Algebraic data types allow for the rigorous definition

of complex types, similarly to an AST definition. They are supported by many functional languages such as OCAML, TOM, etc. The algebraic types in INI differ from typical ones because they allow for named field definitions and because the field initializations are not mandatory when constructing a type. Using an algebraic type, we can specify better the tree structure as follows:

```
type Tree = Leaf[value:Int]
          | Node[value:Int,left:Tree,right:Tree]
```

As we an see, we now have a `Tree` type that provides two constructors: one for regular nodes, and one for leaves. We can slightly change the function implementation:

```
1  function fibtree2(v) {
2    @init() && v>=2 {
3      root = Node[value=v]
4    }
5    @init() && v==1 {
6      root = Leaf[value=v]
7    }
8    n of Node | n.value > 2 && !n.left {
9      n.left = Node[value=n.value - 1]
10     n.right = Node[value=n.value - 2]
11   }
12   n of Node | n.value == 2 && !n.left {
13     n.left = Leaf[value=1]
14     n.right = Leaf[value=1]
15   }
16   @end() {
17     return root
18   }
19 }
```

INI provides a match operator ~ that programmers can use to match instances of algebraic types. When a match is done in the guard, the matched object is considered to be an instance of the type constructed with the constructor used in the match expression. For instance, one can replace "`n of Node | n.value>2 && !n.left`" with "`n of Node | n ~ Node[value>2, !left]`". Using a match operator in a set expression is not very useful, since here, we already know that `n` is an element of the `Node` set. Matching against the `Node[value>2, !left]` expression is thus redundant. However, using

match expression can be useful when programmers want to program using the functional style, for example to define a recursive function with a match switch, like one would proceed in OCAML, for instance. Here is a recursive implementation using functional programming style (note that it does not take a value, but a node `n`):

```
function fibtree3(n) {
  n ~ Node[value==2,!left] {
    n.left = Leaf[value=1]
    n.right = Leaf[value=1]
  }
  n ~ Node[value>2,!left] {
    n.left = fibtree3(Node[value=n.value - 1])
    n.right = fibtree3(Node[value=n.value - 2])
  }
  @end() {
    return n
  }
}
```

Finally, here is the `fibtree4` function, which is a small variation of the `fibtree3` function. `fibtree4` creates new nodes to replace the matched node `n` instead of setting the `left` and `right` fields.

```
function fibtree4(n) {
  n ~ Node[value==2,!left] {
    n = Node[left=Leaf[value=1],
             right=Leaf[value=1]]
  }
  n ~ Node[value>2,!left] {
    n = Node[left=fibtree4(Node[value=n.value - 1]),
             right=fibtree4(Node[value=n.value - 2])]
  }
  @end() {
    return n
  }
}
```

5.4 Programming a simple HTTP server

In order to illustrate the use of Java objets and the use of asynchronous event-based rules, we will now study a slightly more complex example. The

goal of this example is to program a very simple HTTP server, which is not meant to be complete, but to explain basic programming principles in INI. For this example, we will use the `java.net` API for sockets, and some of the `java.io` classes. To use Java objects from INI the programmer just need to define bindings from INI functions to Java constructors, methods or fields. For instance, to bind an `out()` function to the `System.out` Java writer, you need to define the following binding:

```
out()->Writer => "java.lang.System", "out"
```

Note that this binding defines a new function `out`, which is typed in INI with the functional type `()->Writer` as indicated in the binding definition. Since the `Writer` type does not exist in INI, INI will create a new type so that type checking can be performed properly. You can now define a binding for the Java writer's `println(String)` function. Note that since this function is not static, in INI you must pass the instance as the first parameter of the function:

```
java_println(Writer,String)->Void
  => "java.io.Writer", "println(..)"
```

Note that we name the binding `java_println` to avoid the name clash with the `println` built-in INI function. Also, note the `"println(..)"` notation, which means that INI will try to map to the function named `println`, with the best match in the parameter types. Once the bindings have been defined, the programmer can use them from their INI functions. Bindings must always be declared at the beginning of INI files. For instance:

```
1 out()->Writer => "java.lang.System", "out"
2 java_println(Writer,String)->Void
3   => "java.io.Writer", "println(..)"
4
5 function main() {
6   @init() {
7     java_println(out(),"hello Java")
8   }
9 }
```

Type safety within the INI program is ensured through the binding declarations. It is the task of the programmers to define consistent bindings with regards to the types. Finally, the bindings to Java class constructors are achieved by using `"new(..)"` for the method name.

For programming our HTTP server, we will need to define a few bindings to Java functions, which we give right away:

```
socket_server(Int)->ServerSocket
  => "java.net.ServerSocket", "new(..)"
socket_accept(ServerSocket)->Socket
  => "java.net.ServerSocket", "accept(..)"
socket_address(Socket)->InetAddress
  => "java.net.Socket", "getInetAddress(..)"
host_name(InetAddress)->String
  => "java.net.InetAddress", "getHostName(..)"
socket_input_stream(Socket)->InputStream
  => "java.net.Socket", "getInputStream(..)"
socket_output_stream(Socket)->OutputStream
  => "java.net.Socket", "getOutputStream(..)"
reader(InputStream)->Reader
  => "java.io.InputStreamReader", "new(..)"
buffered_reader(Reader)->BufferedReader
  => "java.io.BufferedReader", "new(..)"
data_output_stream(OutputStream)->DataOutputStream
  => "java.io.DataOutputStream", "new(..)"
read_line(BufferedReader)->String
  => "java.io.BufferedReader", "readLine(..)"
write_string(DataOutputStream,String)->Void
  => "java.io.DataOutputStream", "writeBytes(..)"
close(DataOutputStream)->Void
  => "java.io.Closeable", "close(..)"
```

Next, we implement a function `start_http_server` that will use a server socket to listen to some TCP client connections. For those who are not familiar with the TCP socket API, the principles are the following: the programmer must open a server socket and call the `accept` function on it in order to wait for some client connections. The call to `accept` is always blocking, meaning that the program will halt until a connection happens. Once a new connection is made by a TCP client, the programs starts again and the `accept` function returns a new socket on which the client/server communication can be done. Thus the server may continue to listen to some other client connections on the initial server socket, assuming that the program deals with the client requests in different threads or processes.

```
function start_http_server(port) {
  @init() {
    s = socket_server(port)
    println("Server started on "+port)
  }
  s {
    // wait for a connection
```

```
8       c = socket_accept(s)
9       // here we can deal with the connections
10      ...
11    }
12  }
```

In code above, the server socket is constructed line 3. You can note the use of the `s` guard on line 6. This guard allows the `socket_accept` function to be called under the condition that the server socket `s` is actually constructed. Use of such variable existence rules for programming is typical of rule-based programing with INI and allows safer programming since it encourages the programmer to define all the conditions under which actions can be executed.

It would be possible to deal with client connections after the `socket_accept` function call, as indicated at lines 9 and 10. However, several client connections would not be able to be handled in parallel. Thus, with INI, the best way to deal with the connections is to use an event-trigerred rule. Event-trigerred rules are asynchronous and execute in different execution threads and straightforwardly allows for multithreaded programs. Here, we can use the `@update` event on the `c` variable, which will trigger the action when the variable is updated, as the event name says.

```
1  function start_http_server(port) {
2    @init() {
3      s = socket_server(port)
4      clear(c)
5      println("Server started on "+port)
6    }
7    s {
8      c = socket_accept(s)
9    }
10   @update() [variable=c] {
11     // deal with the connections (asynchronous)
12     ...
13   }
14 }
```

Note the `clear(c)` statement added at line 4, which is necessary for the `@update` guard to be correctly installed on the `c` variable. The `clear` function is a built-in function and allows for (re)-initialization of a variable content. It is often used to make sure that the variable is defined when `@update` event is installed, i.e. right after the function's initialization. In

order to deal with the client connections we use the input and output streams of the socket produced by the `socket_accept` function. For simplicity, we delegate to a `handle_http_request` function.

```
1  // Java bindings
2  socket_server(Int)->ServerSocket
3    => "java.net.ServerSocket", "new(..)"
4  ...
5
6  function main() {
7    @init() {
8      start_http_server(8080)
9    }
10 }
11
12 function start_http_server(port) {
13   @init() {
14     s = socket_server(port)
15     clear(c)
16     println("Server started on "+port)
17   }
18   s {
19     c = socket_accept(s)
20   }
21   @update() [variable=c] {
22     client = socket_address(c)
23     println(to_string(host_name(client))
24       +" connected to server")
25     in = buffered_reader(reader(
26       socket_input_stream(c)))
27     out = data_output_stream(socket_output_stream(c))
28     handle_http_request(in, out)
29   }
30 }
31
32 function handle_http_request(in, out) {
33   line!="" {
34     line = read_line(in)
35   }
36   line ~ regexp("GET (.*) (.*)", path, version) {
```

```
37      println("GET FOUND: "+path+" - "+version)
38    }
39    @end() path {
40      // generate an empty response
41      write_string(out, "HTTP/1.0 200 OK\r\n")
42      write_string(out, "Connection: close\r\n")
43      write_string(out, "Server: INI v0\r\n")
44      write_string(out, "Content-Type: text/html\r\n")
45      write_string(out, "Content-Length: 0\r\n\r\n")
46      close(out)
47      println("end of request")
48    }
49    @end() !path {
50      close(out)
51      println("illegal request")
52    }
53  }
```

For dealing with HTTP requests, we use the match operator for strings, which allows matching against regular expressions (at line 36). The regular expression seeks for a `GET` HTTP command. In the HTTP protocol syntax, the `GET` command is followed by the path and the HTTP version, which are space-separated. In the regular expression [2] standard used by INI (see [...]), parenthesis produce match outputs, which can be bound to INI variables. Here, `regexp("GET (.*) (.*)", path, version)` binds the two space-separated expressions following the `GET` command to the `path` and `version` variables. This binding enables the `@end()` event to trigger the code that generates the HTTP response page, since the guard is only true upon the `path` condition at line 39. In the case the `path` variable is not bound, the function ends without responding to the request (line 50).

Finally we can add a function to generate some HTML content. In INI, one can reference functions and pass them as parameters to other functions (similarly to closures in functional languages). Here, for instance, we can make the program more generic by passing a function that will handle the requests. Thus, we can add a `handler` functional parameter to the `start_http_server` function.

```
1   [...]
2
3   function start_http_server(port,handler) {
4     @update() [variable=c] {
```

```
5       [...]
6       handle_http_request(in, out, handler)
7     }
8   }
9
10  function handle_http_request(in, out, handler) {
11    [...]
12    @end() && path {
13      body = eval(handler,path)
14      write_string(out, "HTTP/1.0 200 OK\r\n")
15      write_string(out, "Connection: close\r\n")
16      write_string(out, "Server: INI v0\r\n")
17      write_string(out, "Content-Type: text/html\r\n")
18      write_string(out, "Content-Length: "+
19        size(body)+"\r\n\r\n")
20      write_string(out, body)
21      close(out)
22      println("end of request")
23    }
24    [...]
25  }
```

The function referenced by the `handler` variable is evaluated at line 13, using the `eval` built-in function. The `eval` function takes a function as a parameter and the parameter list to be passed to the evaluated function. It returns the result of the evaluated function. In terms of typing, the inferred type of `eval` depends on the inferred type of the function to be evaluated: `eval(f:F,param_list):(F,T1,T2,...Tn)->T`, where `F=(T1,T2,...Tn)->T`. Here, we expect the `handler` function to take a string for the path of the requested URL and to return a string that will be used as the body of the HTTP response. For instance, in the following code, we can define a `my_handler` function that generates a "hello world" HTML page. Note the use of the `@init()` and `@end()` rules, which can straightforwardly be used to wrap the response into an HTML body. Also, rules can be used to generate different HTML pages depending on the URL's path. Here, `my_handler` will return a "hello world" content for the `test` path (line 5), but an empty content for other paths. Note also the use of the `clear(path)` statement of line 7 to avoid the infinite application of the rule. One could also change the path value (or other variable value) in the rule to trigger another rule.

```
1  function my_handler(path) {
2    @init() {
3      s = "<html><body>"
4    }
5    path == "/test" {
6        s = s + "hello world"
7        clear(path)
8    }
9    @end() {
10     s = s + "</body></html>"
11     return s
12   }
13 }
```

Finally, this rule-based function looks similar to a controller in the MVC architecture [...], except that the conditions that trigger the actions are clearly specified in the code. Conditions may hold on the request state (path, parameters, session variables), and possibly on the server state, even though is would probably not be a nice idea to program a statefull HTTP server. To make our server more complete, we should at least deal with the GET parameters and the POST requests and parameters. For sake of simplicity, we will not enter in such details here.

Finally, we can start our HTTP server by passing the right handler function. Note the use of the `function` keyword at line 3 in the `main` function, which allows the program to get a function reference on the function named `my_handler`.

```
1  function main() {
2    @init() {
3      start_http_server(8080, function(my_handler))
4    }
5  }
```

6 Candidate features

Following features are not yet implemented but could be in future releases, especially if people up-vote them on github...

6.1 Validations and Model Checking

In the close future, we want to add to INI some validations to avoid programming mistakes (besides those already detected by the type inference algorithm). Validations will be performed with a model-checking engine called Spin. The idea is to provide default validations such as:

- Detection of accesses to undefined variables. This should be pretty easy. We can only perform local analysis for each rule. The analysis would verify that the guard ensures the used variables existence. For instance `a{println(a)}` will be correct, whereas `a{println(b)}` will not (unless the analysis can decide that `b` has been initialized by some other means).
- Type fields should be mandatory by default. Maybe a syntactic construct should be introduced for optional fields.
- Detection of concurrent variable accesses (maybe solved by some embedded transactional construct)

Besides default validations, programmers will be able to define their own using Temporal Logic formulas.

6.2 Interceptors

Interceptors would allow to react upon some specific function invocation or process spawning. For example:

```
@before(function(myfunction),args) {
  // do something
}
@after(function(myfunction),args,result) {
  // do something
}
@around(function(myfunction),args) {
  // do something
  result = proceed(function(myfunction), args)
  // do something
}
```

6.3 `@file` event

Observing file system modifications can be useful in many use cases.

The `@file(f1:String, f2: String, ... fn:String)` event occurs when one of the given file `fi` is modified on the file system. If the file does not exist initially, the event occurs when the file is created. If the file initially exists, the event occurs when it is modified or deleted. If the given file is a directory, the event happens when the content of the directory is modified (i.e. a file within this directory is modified, created, or deleted).

6.4 Date and Duration type

Add a built-in Date type with + and - operators. Date literals will need to be specified, as well as date formatting. For type consistency, dates may be seen as durations from the initial date.

References

[1] W. R. Franta and Kurt Maly. An efficient data structure for the simulation event set. *Commun. ACM*, 20(8):596–602, 1977.

[2] Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

[3] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.