INI Language Specifications

Renaud Pawlak, CINCHEO

Version pre-alpha 2

** IMPORTANT WARNING ***
This document is not up-to-date (yet) w.r.t. the current implementation.
Please contact the author for full details and refer to the GitHub project.
`https://github.com/cincheo/ini`

Contents

Introduction

INI is a programming language dedicated to distributed computing and pipeline creation. It natively handle processes, communication, deployment and synchronization.

The goal of this document it is provide detailed insight on the INI language, in terms of syntax, typing, model-checking, and semantics. This document is not a tutorial or a simple way to get started with INI.

For a global overview and a motivation statement, as well as to get started with INI, please visit: `https://github.com/cincheo/ini`.

Chapter 1

Language features

1.1 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"`

1.2 Functions

1.2.1 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> }
}
```

1.2.2 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`.

1.2.3 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.

1.2.4 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.

1.2.5 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.

- `any()->Any`: returns a variable that is empty (not initialized) and without any typing constraints.
- `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.
- `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.
- `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.
- `print(v:T)->Void`: writes the textual representation of the passed data on the standard output stream.
- `first(list:T*)->T`: returns the first element in the given list.
- `key(dict:Dictionary(K,V), element:V)->K`: returns the key that corresponds to the given element in the given dictionary.
- `max(n1:Number, n2:Number)->Number`: returns the highest number between `n1` and `n2`.
- `min(n1:Number, n2:Number)->Number`: returns the lowest number between `n1` and `n2`.
- `parse_number(n:String)->Number`: parses the given string and returns it as a number.
- `pow(n1:Number, n2:Number)->Number`: returns `n1` to the power of `n2`.
- `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.
- `produce(channel:String, data:Any)->Void`: produces the given data on the given channel (non-blocking call).
- `read_keyboard()->String`: reads a line typed in by the user on the standard input stream and returns it.
- `size(Dictionary(K,V))->Int`: returns the size of a dictionary, i.e. the number of elements in it.
- `sleep(time:Long)->Void`: stops the current thread during `time` milliseconds.
- `stop(rule:Any)->Void`: stops the given rule, so that it will kill the current thread and terminate the rule (an event rule will not be triggered anymore once stopped).

- `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_json(v:T)->String`: converts the given variable to a String value holding the JSON representation of the data (serialization).
- `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.

1.2.6 Bindings to external functions

INI only provides a minimal set of built-in functions. For all other functions, one can bind new functions (typically to Java APIs, which are available from the default implementation). 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:

```
declare <name>(<types>) => <type> [class="string1", member="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 given annotation, 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 [class="java.lang.System", member="out"]
java_println(Writer,String)=>Void [class="java.io.Writer", member="println(..)"]
```

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

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

1.3 Rules

1.3.1 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.

1.3.2 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 process:

```
process ping_pong() {
   @init() {
    v = any()
   }
  !v {
    println("ping")
    v=""
  }
  v {
```

```
10      println(”pong”)
11      clear(v)
12    }
13  }
```

This process 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:

```
process ping_pong() {
   @init() {
    v = any()
   }
  v {
    println(”pong”)
    clear(v)
  }
  !v {
    println(”ping”)
    v=””
  }
}
```

1.3.3 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.

1.3.4 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.

1.3.5 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 process thread), but some event are fired asynchronously and concurrently (multi-threaded evaluation).

1.4 Synchronous events

Synchronous events are evaluated within the process evaluation thread. When a synchronous event evaluates, no other rule or event can be evaluated concurrently.

1.4.1 `@init` event

As the name says, `@init`-trigger rules are invoked when the process 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:

```
process 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 process evaluation and print out a message:

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

1.4.2 `@end` event

On the contrary to the `@init` event, `@end` is triggered when no more rule apply, and when the process 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.

1.4.3 `@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`.

The `@update` event is particular because it is evaluated synchronously by default, but it can be set to be asynchronous, as explained later.

1.4.4 `@error` event

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

1.5 Processes

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

1.5.1 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:

```
process p() {
 @every() [time=1000] {
  println("hello")
 }
}
```

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:

```
process main() {
 @init() {
  p()
 }
}
```

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)
 }
}
```

1.5.2 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)
 }
}
```

1.5.3 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:

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

1.6 Asynchronous events

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

1.6.1 `@every` event

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

1.6.2 `@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.

1.6.3 `@consume` event

The `@consume(value:Any) [channel = name:String]` event occurs when a data is consumed from the channel `name`. The consumed data is then passed in the `value` variable.

1.6.4 `@update` event (asynchronous mode)

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`.

The `@update` event is particular because it is evaluated synchronously by default, but it can be set to be asynchronous, by setting the optional `mode` annotation parameter to `"async"`. For instance: `@update(b,c) [variable = a, mode = "async"]`.

1.7 Synchronization of events

Since asynchronous events can run concurrently, it is sometimes necessary to use synchronization to avoid undesirable side effects. INI provides a simple synchronization construct with the following syntax, which just complements the declaration of a rule: `$(r1, r2, ... rn) <guard>` { `<statements>` }. It means that the declared rule cannot run if one of the `r1, r2, ... rn` rule is executed.

Rule names are defined trough labels, for example, to name a rule `r`, just add the label in front of the rule as: `r:<guard>` { `<statements>` }. For example, if we have a `@consume` event rule that we want to be executed only once at a time, then we can just write `$(c) c:@consume(v)` { `<statements>` }, so that the event rule `c` synchronizes on itself.

In the following example, the two rules here are made to be mutually exclusive so that the value of the `tick` variable cannot be modified concurrently and remains consistent within the every and consume events.

```
process main() {
  @init() {
    tick = 0
  }
  $(c) e:@every() [time=1000] {
    tick++
  }
  $(e) c:@consume(v) [channel="c"] {
    println("new event at tick = "+tick)
    tick=0
  }
}
```

1.8 Dictionaries, lists, and sets

1.8.1 Dictionary definition and access

Dictionaries are natively supported by INI. A dictionary is automatically defined when accessed through the dictionary access expression that uses square brackets: `dict[key]`. Keys and values within dictionaries are of any type, but shall all be of the same type for a given dictionary. 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 dictionary 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 dict[”key1”] = 1
2 dict[1] = 2
3 dict[”key2”] = ”test”
4 println(dict[”key2”])
```

Note that the size of a dictionary (i.e. the number of elements in it) can be retrieved with the `size` built-in function (see Section 1.2.5). Emptying a dictionary or removing an entry is done with the `clear` function.

1.8.2 List definition and access

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

1.8.3 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]`.

1.8.4 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 [...]).

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

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.

1.9 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.

1.9.1 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.

1.9.2 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)`:

```
expr = UMinus[operand=Plus[
  left=Mult[
    left=Number[value=3.0],
    right=Number[value=2.0]],
  right=Number[value=1.0]]]
```

1.9.3 Type set selection expressions

In Section 1.8.4, 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.

1.9.4 Type set match expressions

In Section 1.3.4, 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    expr = UMinus[operand=Plus[
10     left=Mult[
11       left=Number[value=3.0],
12       right=Number[value=2.0]],
13       right=Number[value=1.0]]]
14   println("The value of "+expr_str(expr)+" is "+expr_val(expr))
15 }
16
17 function expr_str(expr) {
18   case {
19     expr ~ Number[value] {
20       return to_string(expr.value)
21     }
22     expr ~ Plus[left,right] {
23       return "("+expr_str(expr.left)+"+"+expr_str(expr.right)+")"
24     }
25     expr ~ Mult[left,right] {
26       return "("+expr_str(expr.left)+"*"+expr_str(expr.right)+")"
27     }
28     expr ~ Minus[left,right] {
29       return "("+expr_str(expr.left)+"-"+expr_str(expr.right)+")"
30     }
31     expr ~ Div[left,right] {
32       return "("+expr_str(expr.left)+"/"+expr_str(expr.right)+")"
33     }
34     expr ~ UMinus[operand] {
35       return "-("+expr_str(expr.operand)+")"
36     }
37   }
38 }
39
40 function expr_val(expr) {
41   case {
42     expr ~ Number[value] {
43       return expr.value
44     }
45     expr ~ Plus[left,right] {
46       return expr_val(expr.left)+expr_val(expr.right)
47     }
48     expr ~ Mult[left,right] {
      return expr_val(expr.left)*expr_val(expr.right)
    }
    expr ~ Minus[left,right] {
      return expr_val(expr.left)-expr_val(expr.right)
    }
    expr ~ Div[left,right] {
      return expr_val(expr.left)/expr_val(expr.right)
    }
    expr ~ UMinus[operand] {
      return -expr_val(expr.operand)
    }
  }
}
```

1.10 Lambdas

Functions can be assigned to variables and passed as parameters to other functions.

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

1.11 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.

1.12 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.

```java

package ini.ext.events;

import ini.ast.AtPredicate;
import ini.ast.Rule;
import ini.eval.IniEval;
import ini.eval.at.At;
import ini.eval.data.Data;
import ini.eval.data.RawData;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.HashMap;
import java.util.Map;

public class AtCopyFile extends At {
  public String sourceFileName;
  public String destFileName;
  public long copyTime;
  public boolean success = false;
  Map<String, Data> variables = new
        HashMap<String, Data>();
  Map<Thread, At> threadAt = new HashMap<Thread, At>();
  @Override
  public void eval(final IniEval eval) {
    final Data sfn = getInContext().get("source");
    sourceFileName = (String) sfn.getValue();
    final Data dfn = getInContext().get("destination");
    destFileName = (String) dfn.getValue();
    new Thread() {
      @Override
      public void run() {
        try {
          long beginTime = System.currentTimeMillis();
          File sourceFile = new File(sourceFileName);
          File destFile = new File(destFileName);
          InputStream in = new
                         FileInputStream(sourceFile);
          OutputStream out =
                         new FileOutputStream(destFile);
          byte[] buffer = new byte[1024];
          int fLength;
          while ((fLength = in.read(buffer)) > 0) {
            out.write(buffer, 0, fLength);
          }
          in.close();
          out.close();
          long endTime = System.currentTimeMillis();
          copyTime = endTime - beginTime;
          success = true;
          variables.put(getAtPredicate().outParameters.
                              get(0).toString(),
              new RawData(copyTime));
          variables.put(getAtPredicate().outParameters.
                              get(1).toString(),
              new RawData(success));
          this.setName("Demo copy thread");
          execute(eval, variables);
          terminate();
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }.start();
  }
}
```

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

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

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.

Chapter 2

The INI type system

2.1 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 dictionary types: $Dictionary(K, V)$. Dictionary 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:

$$T * \widehat{=} Dictionary(Int, T)$$

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).

$$String \widehat{=} Char * \widehat{=} Dictionary(Int, Char)$$

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.

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

2.2 Type Inference

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 [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.

2.3 Type Inference rules

2.3.1 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`,

2.3.2 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.

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

2.3.3 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.

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

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

2.3.4 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.

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

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

2.3.5 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.

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

2.3.6 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.

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

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

2.3.7 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.

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

2.3.8 The list concatenation operator (TBD)

The list concatenation operator is `&`.

$$\text{(CONCAT)}\ \frac{\vdash x : T* \qquad \vdash y : T*}{\vdash (x\ \texttt{\&}\ y) : T*}$$

2.3.9 Unary minus (sign inversion)

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

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

2.3.10 Post operators

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

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

with `postop` $\in$ {`++`, `--`}

2.3.11 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).

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

2.3.12 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.

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

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.

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

2.3.13 Dictionary access

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

$$(\text{DICTACCESS})\ \frac{\vdash x : \mathit{Dictionary}(K, V) \qquad \vdash y : K}{\vdash x[y] : V}$$

2.3.14 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.

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

2.3.15 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.

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

2.3.16 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 `Dictionary(Int,T)`)

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

2.3.17 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.

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

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

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

2.3.18 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

2.3.19 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)`.

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

2.3.20 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.

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

Bibliography

[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.