IAS 0.3

(Interaction Server)

Gil Müller, gil.mueller@nexgo.de

12.5.2014

1. Introduction

IAS (Interaction Server) is a server, which is controlled by a script. The server is designed to be used for protocol handling and streaming. Most suitable would be a role as an intermediate. Additionally, it may be used for parsing and file conversion.

The current version, which is described in the document, is very rudimentary. Its sole purpose is to serve as a starting point for the concept.

As you will notice, the language of IAS is not entirely new. It is rather a mixture of ingredients taken from languages like Smalltalk, Scheme und the C family of languages (C, C++ and Java). Still, this a new language, which will you particularly recognize if you look at the block concept, which serves both for data and procedural abstraction. Additionally, it hosts a meta-interpreter, which eases the parsing and interpretation of files and streams.

Particularly, the meta-interpreter supports the interpretation of domain specific languages. It directly evaluates syntactic patterns, so that the processing is as natural as writing a function. For more complex languages several layers can be used. Additionally, it also supports a feature-oriented language design [1].

2. Usage

IAS is called as follows:

```
python ias <defs> <script> <args>
```

defs: a list a definitions; each definition has the form: <name>=<value>); name is a valid name in the language of IAS (cf. below) and value is a string value (e.g. foo=bar); the list is included in the root environment.

script: a script in the language of IAS

args: any number of command line arguments, which are available in IAS using the argv name.

Note, that IAS was tested with Python 2.5.

3. The Language of IAS

The behavior of IAS is controlled by a script. In this chapter the script language is explained.

3.1 Basics

A script is a text file with any number of lines. A comment starts with # and ends at the end of the line:

```
# this is a comment
```

All other text is interpreted as actions. An action influences the behaviour of ias and has always an outcome: the result. The simplest actions are constants.

3.1.1 Nil

There is the nil constant, which is a placeholder for a value without meaning.

3.1.2 String

String constants are enclosed in double quotes. E.g.: "a string". Some characters may be escaped:

\\ (backslash)

\" (double quote)

\n (linefeed)

\r (carriage return)

\t (tab)

For example: `"\\\""` yields the string \"

The following operations are supported:

+ operator: concatenation of strings; `"a" + "b"` yields "ab"

size(aString): returns the string length; `size("abc")` yields 3

[] operator: accesses a component in the string; `s="abc"; s[0]` yields "a"

substring(aString, start, end): returns a substring of the given string starting at start and ending at the character before end; `substring("abc", 1, 3)` yields "bc".

string(someValue): tuns the argument into a string

3.1.3 Int

For integers the following operations are available:

+ (Addition): 1 + 2 yields 3

- (Subtraction): 1 - 2 yields -1

int(value): converts a value to an integer (most useful if the value is a string)

Note, that negative integers - at that point – are only possible using subtraction: `0 - 1` yields -1.

3.1.4 Bool

The boolean constants are true and false. The following operations are provided:

! (Not): `!true` yields false

&&(And): `true && false` yields false. Note, that if the first argument is false, the second argument is not evaluated.

|| (Or): `true || false` yields false. Note, that if the first argument is true, the second argument is not evaluated.

3.1.5 Vector

Vectors are containers of ordered values.They can have any length. Use the following operations:

vector(arg1, ..., argn): generates a vector with the given arguments as its components

allocateVector(size, initialValue): generates a vector of the given size and with the given initialValue as the value of all components

+ (Concatenation): `vector(1) + vector(2,3)` yields vector(1,2,3)

size(vector): the length of the vector; `size(vector(1,2))` yields 2

[] (Access or assignment): `vector(1,2)[1] yields 2. v=vector(1,2); v[0]=3; v` yields vector(3,2)

3.1.6 Symbol

A symbol is a name, which can be processed. A symbol is created by stating a dollar sign and the name for the symbol. E.g.

```
$n1
```

creates the symbol with name n1.

3.1.7 Comparism

You may compare values of the same type. The exception is to compare a value with nil:

== (Equal): `1 == 1` yields true; `1 == nil` yields false

!= (Not Equal): `1 != 1` yields false

< (Smaller): `"asd" < "bsd"` yields true

<= (Smaller or Equal): `"qwe" <= "qwe"` yields true

3.1.8 Sequence

You may perform a sequence of actions by using a semicolon:

```
 "one"; "two"
```

While all actions in the sequence are executed. the outcome is always the last action in the sequence (in the example we would get "two").

3.1.9 If

An if action has the following form (the else part is optional):

```
if <condition>
  <then-part>
[else
  <else-part>]
```

If the condition yields true, the then-part is executed; if the condition is false the else-part is executed. The result of the if action is either the result of the executed part or nil if no part was executed. E.g.

`if true 1 else 2` yields 1

`if false 1` yields nil

`if (1+1 < 2) 2 else 3` yields 3 (note: for complex expressions you have to use parentheses)

3.1.10 While

A while action has the following form:

```
while <condition>
  <body>
```

At first the condition is executed. If the result is true the body is executed. This is repeated as long as long as condition is true. The result of while is the result of the last result of the executed body or nil when the body was never executed.

For example:

`while false 1` yields nil

`v = vector(): while (size(v) < 3) v = v + vector(1); v` yields vector(1,1)

3.2 Names

Names are used as references to values.

A simple name consists of any number of roman letters, digits and underscores, but may not start with a digit. The following examples show valid names: aName, _someName2, and_one_more_name.

The following names have a special meaning and may not be assigned to a value (cf. below):

this – the current environment

super – the environment which is above the current environment (is bound to nil for the top level environment).

There are three ways to assign names to values

1. by assign: using the assign action or the conditional assign action:
   ```
   a = "1" # assigns the string to a in any case
   a ?= "" # assigns the string only to a if has no value yet
   ```
2. by default: `{ a= nil : nil } # a is bound to nil when the block is evaluated (cf. below)`
3. by application `{ a : a }("1") # a is bound to "1" when the block is executed (cf. below)`

You may compose names in order to refer the values which are bound in an environment. Composed names are separated by dots. E.g.: some.special.name

If a composed name starts with extern, it refers to values in the python runtime enviroment. For example: extern.id

Composed names are explained in more detail in chapter 3.3.

3.3 Blocks

3.3.1 Form and Evaluation

A block is a parameterized composed action. It has the following form { <parameters> : <an action> }. Parameters is a list of (simple) names, which may be assigned a default value. E.g. `{ a, b = "c" : a }` specifies a block with two parameters a and b. The default value of b is "c". The action of the block is the reference to a.

If no default value is given for a parameter it is assigned nil.

If a block is evaluated the result is a closure, which a special value, which contains a reference to the environment, in which the block was defined. For example:

```
outerF = {
    innerF = { a, b = "c" : a } # f is assigned the closure of the block
                                # with a reference to outerF
}
```

3.3.2 Application

Blocks may be executed by applying them to a list of values. First a new environment is created, whose parent is the environment of the closure. Next, the parameters are bound in the new environment to their default values. Then, the given arguments are assigned to the parameters in the order as they were defined. Finally the action of the block is executed.

E.g. `f ("1", "2") # the result is "1" (a is bound to "1" and b is bound to "2")`

You may also partially apply values to the f block:

```
f () # the result is nil (a is bound to the default value of nil)
```

You may also apply values to a block using the star application:

`f * (vector("1", "2"))` yields again "1"

The argument behind * has to be a vector. The components of the vector become the argument of the block.

3.3.3 Objects

Objects are environments which become data objects in IAS. Blocks may also be used to create objects:

```
o = { a= "1" : this } () ; # creates an object
o.a # the result is "1"
```

The following operations are supported:

size: returns the number of bindings in the object. E.g. `size({a: this}(1))` yields 1

== (Equal): is true if the given arguments refer to the same object

!= (Not Equal): is true if the given argument refer not to the same object

[] (Access and assignment): `o={a=1}(); o["a"]` yields 1; `o={a=1}();o["a"]=2; o.a` yields 2

bound(object, name): checks if the name is bound in the object (or in any predecessor)

lookup(object, name): retrieves the value bound to name.

includeBindings(destination, source): includes the bindings from object source in object destination.

3.3.4 Name Resolution

To refer to a name (either by using it or assigning a value to it), the environment needs to be located, in which the name is defined. For a simple name a bottom-up search is performed: the search goes upward until an environment is found, where the name is defined. The same happens for the first component in a composed name. The other components will be resolved just in the environment to which the previous component is bound.

Both references (the last two lines) in the next example refer to the same definition (and value):

```
a=1;
{ super.a } ();
{ a} ()
```

If is possible to refer to all names in an environment using a composed name which ends with *. E.g. `o = { a=1; b= 2; this }() ; o.*` yields vector("a", "b")

This means * refers to a vector of all names bound in the environment.

If you use a simple name on the left hand side of an assignment, the name is bound (or rebound) in the current environment. Thus, if you like to change (or add) the binding in a different environment, you have to refer to it. E.g.

```
  a=1;
  f = {
    a = 1;  # assignment happens in environment of f
    super.a = 1 # assignment happens in environment of top level
 }
```

3.4 Streaming

IAS has a construct called transfer in order to simplify streaming. It is used as follows:

```
inGate > outGate
```

Both arguments have to be gates. A gate is basically a communication endpoint. The transfer action takes data from inGate and transport it to outGate. You may also chain transfer – if the gate supports reading and writing:

```
gate1 > gate2 > ... > gateN
```

Currently the following gate types are built in (given by their generator functions):

constGate(value): provides the given value

vectorGate(vector): provides the components of the given vector

fileGate(name, mode): reads or writes a file (cf. the mode parameter)

consoleGate(): reads or writes from the console (reading is line-oriented!)

socketSrvGate(localPort): reads and writes from a server socket

socketClntGate(remoteHost, remotePort): writes and reads from a client socket

The gates are part of the io module, so that you have to access them e.g. like io.constGate.

The names stdin and stdout refer initially to consoleGate (cf. init.ias)

3.5 The Meta-Interpreter

The module mi includes a meta interpreter. It provides the following functions:

interpret(vectorOfModules, aString): the given string is parsed and interpreted using the vector of modules (cf. below). The result is the result of the start rule.

interpeterGate(vectorOfModules): returns a gate, which parses and interprets its input. The result of the interpretation will be the output of the gate (e.g. stdin > interpreterGate(modules) > stdout).

The basis for the interpretation is the vector (or stack) of modules. Each module is a block, which contains at least two bindings:

start - the name of the start rule

rules - a block of an object which contains the syntactic (and semantic) rules. A rule is a binding consisting of the name of the rule and a block of an object containing the alternatives of the rule. An alternative has to be a function. Its parameters are the list of terminals and nonterminals. In its body it may process the parameters. There are the following variants for a parameter:

Parameter has default value:

If the default value is a string, then it is considered as a terminal (a regular expression), which is matched with the input. The result is bound to the parameter (overwriting the default value)

If the default value is a symbol, then the default value is taken is a non-terminal (cf. below for the processing). The result is bound to the parameter.

Parameter has no default value and starts with a capital letter: This is a special terminal. The input has to be composed of tagged tokens. The name of the terminal has to match the name of current tagged token in the input. The argument of the token if bound to the parameter.

Parameter has no default value and starts with an underscore (_): The parameter is considered as a generic parameter. This will instruct the interpreter to lookup the value for the parameter (in the environment of the rule). The value is treated like a default value and is processed accordingly (cf. above).

Parameter has no default value and starts with a small letter: This is interpreted as a nonterminal. The interpreter will look for a rule with the same name and proceeds the matching there. The parameter will be bound to the interpretation result of the rule. If the name of the parameter is "self", this is interpreted as a reference to the rule in which the parameter is defined.

If you would like to use several nonterminals with the same name, you may append a digit after the name. The interpreter will look for the rule without the digit (e.g if will look for exp if the nonterminal is specified as exp1).

Consider the following example, which contains a module for lexical analysis ("Lexer") and another one for syntactic analysis ("Parser"). The result will be a count of the ones in each word. For the sake of the example, the Lexer module will encode the string input as a stream of TaggedToken objects (cf. output). The Parser module will analyse the incoming stream and report the count of ones in each word:

```
lexer = {
  start = "start",
  output = mi.TaggedTokenCollection(),
  rules = {
    start = {
      st1 = { symbols : output }
      : this
    },
    symbols = {
      a1 = { binaryDigit, symbols : nil },
      a2 = { stop, start : nil },
      a3 = { nil }
      : this
    },
    binaryDigit = {
      b1 = { d="0" : output.add(mi.TaggedToken("Zero", nil)) },
      b2 = { d="1" : output.add(mi.TaggedToken("One", nil)) }
      : this
    },
    stop = {
      s1 = { d=" " : output.add(mi.TaggedToken("Stop", nil)) }
      : this
    }
    : this
  }
  : this
};

parser = {
  start = "start",
  rules = {
    start = {
      s1 = { fullWord, start : fullWord + start },
      s2 = { "" }
      : this
    },
    fullWord = {
      w1 = { word1, Stop : "Count: " + string(word1) + "\n" }
      : this
    },

    word = {
      w1 = { digit, word : digit + word },
      w2 = { 0 }
      : this
    },
   digit = {
       w1 = { Zero : 0 },
       w2 = { One : 1 }
       : this
     }
     : this
  }
  : this
};
io.constGate("01 1010 111 ")
> mi.interpreterGate(vector(lexer, parser))
> stdout
```

The next example how to use generic rules and how to separate rules. The intepreter accepts additions. The module numberModule contains the rules for the numbers and the supported operation. Note, that the rule add is generic to leave it open what kind of operand can be used. In the main module, the rules are gathered. There, we resolve the rule add so that number is the operand in add. Note, that numberModule has a free form, while mainModule has to conform to the structure which is used by the meta-interpreter:

```
numberModule = {
  add = { _operand :
            { a1 = { _operand, l="\\+", self : _operand+self },
              a2 = { _operand, l=";" : _operand }
              : this }};
  number = { n1 = { l="[0-9]*" : int(l) } : this };
  this
}();

mainModule = {
  start = "start",
  rules = {
    includeBindings(this, numberModule);
    start = {
      st1 = { exprs : exprs }
      : this
    };
    exprs = {
      es1 = { addexpr, self : "sum: "+string(addexpr)+"\n"+string(self) },
      es2 = { addexpr : "sum: " + string(addexpr)+"\n" }
      : this
    };
    addexpr = add($number);
    this
  }
  : this
};

io.constGate("1+11;34+21;") > mi.interpreterGate(vector(mainModule)) > stdout
```

The meta-interpreter works as follows. For each module it takes the output of last module as input (or a string collection of the given string input). It will begin the interpretation with the start rule. It will try to match any of the alternatives in the given order.

An alternative is matched, if all parameters can be matched (for a match of a parameter: cf. above). If the matching succeeds, the alternative is called with the matched input. If it fails, the next alternative is tried. If no further alternative exists, the interpretation of the rule fails.

If the interpretation of a module fails (either the start rule fails or the input was not fully consumed), the interpretation will be aborted.

Other useful definitions in mi (cf. mi.ias):

TokenCollection: hosts the data which serves as input (and output) to a module in the meta-interpreter

StringTokenCollection(aString): creates a collection which contains the given string. Such a collection is typically be used as in an input for the first module.

TaggedToken(name, arg): creates a tagged token.

TaggedTokenCollection(): creates a collection which contains tagged tokens. This collection serves well for processing in later modules (second and higher).

3.6 Other Functionality

There are the following additional functions:

import(filenameWithoutExtension): imports the interpreted content of the file. The result is an object. E.g. use it as follows:

```
myModule = import("myModule");
myModule.someValue
```

regexCompile(pattern): compiles a regular expression and returns the result (is nil in case of an error). Use it together with regexMatch.

regexMatch(compiledPattern, aString): returns a match of the compiled pattern in aString starting at 0. The result is -1 when there was no match or the index after which the match ended. E.g.

```
compiledPattern = regexCompile("[0-9]*");
regexMatch(compiledPattern, "0123wer")
```

yields 4

parameters(aClosure): returns a vector of pairs of the closure's parameters. Each pair consists of the name of the parameter and its default value.

environment(aClosure): returns the environment of the closure.

trace(level, msg): prints a debugging message (level has currently no use). The parameter msg has to be a string.

combine1(f1, f2): combines the given functions. E.g. combine({x: x+1}, {x: x+2})(0) yields 3

map(f, aVector): maps all components of aVector to f and returns the list of results. E.g.

map({x: x+1}, vector(1,2,3)) yields vector(2,3,4)

applyAll(f, aVector): the same as map except there is no result returned.

typeof(item): returns the type of item as a symbol

3.7 Future Work

The next step will be an improvement of the current functionality:

1. Complete the language in terms of basic datatypes and operations; add the * attribute to parameters (as in Python); the usage of names maybe needs some refinement (it is quite complex now).
2. Make parsing simpler
    - Add +, *, ? as defaults for nonterminals (EBNF syntax)
    - Make interpreter more robust (currently the interpreter might freeze if the input is not correct)
    - Provide meaningful error messages (including column and line or error)
3. Make streaming more powerful: the current concept seems too simple. Partly, this relates to adding output in the modules of meta-interpreter. Another part relates to the current concept: the transfer operator controls the streams, which means the gates react to its calls. They have no means to generate output when they receive input. Either this is enabled or a sort of suspend is provided (e.g. using continuations or a coroutines), which is activated when the output is requested.

4. Appendix

The notation for the syntax uses a mixture of BNF (Backus Naur Form) and regular expressions. Multiplicities are notated with the symbols of regular expressions. Note, that regular expressions are also used for the specification of terminals.

3.7.1 Microsyntax

```
iaslang = name | int | string | comment | ";" | "," | "(" | ")" | "{" | "}"
        | "[" | "]" | "==" | "=" | "?=" | ">" | ":" | "." | "&&" | "||"
        | "!=" | "!" | "+" | "-" | "*" | "<=" | "<" | "$"
name = "[a-zA-Z_][a-zA-Z0-9_]*"
int = "[0-9]+"
string = "\" "(\\\"|\\\\|\\n|\\r|\\t|[^\"])*" "\""
whitespace = "\s"
comment = "#[^\n]*\n?\r?"
```

3.7.2 Macrosyntax

```
iaslang = action?

action = sequence
sequence = control sequenceRest*
sequenceRest = ";" control

controlAction = ifAction | whileAction | assignment

ifAction = "if" basicAction assignment elseAction?
elseAction = "else" controlAction
whileAction = "while" basicAction assignment

assignment = assignmentPrefix? transfer
assignmentPrefix = composedName access* assignmentOperator

transfer = orTerm transferRest*
transferRest = ">" orTerm

orTerm = andTerm orTermRest*
orTermRest = "||" andTerm

andTerm = notTerm andTermRest*
andTermRest = "&&" notTerm

notTerm = "!" compareTerm | compareTerm

compareTerm = arithmetic compareTermRest?
compareTermRest = compareOperator arithmetic

arithmetic = simpleAction arithmeticRest*
arithmeticRest = arithmeticOperator simpleAction
```

```
simpleAction = basicAction applicationOrAccess*
applicationOrAccess = listOfActions | "*" basicAction | access

basicAction = parenAction | constant | block | composedName
parenAction = "(" action ")"

constant = string | "nil" | "true" | "false" | int | "$" name
block = "{" defs? action "}"
access = "[" action "]"
listOfActions = "(" actionSequence? ")"
actionSequence = action ("," action)*

defs = names ":"
names = nameDeclaration ("," nameDeclaration)*
nameDeclaration = name ("=" simpleAction)?
composedName = name composedNameRest* composedNameAll?
composedNameRest = "." name
composedNameAll = "." "*"

assignmentOperator = "=" | "?="
compareOperator = "==" | "!=" | "<" | "<="
arithmeticOperator = "+" | "-"
```

4. References

[1] G. Müller: "*Modular Programming Language Design*", http://www.gil-mueller.com/ias/MPLD.pdf, 2014.