ATOM Documentation and Specifications

January 4, 2018

1 Introduction [intro]

1 This document specifies implementation of ATOM language, yet it serves as documentation of *the* implementation as well. For this reason, this document will have two kinds of specifications, namely clauses and implementation details, which are applicable to the ATOM language and this particular implementation, respectively.

2 ATOM is general purpose equation-based physics engine (IVP solver) generator focusing on generating efficient code in a user friendly approach with great flexibility.

2 Terms [intro.term]

1 This section defines terms used in this documentation that have different meanings than in other context or need to be clarified for potential ambiguity. Refer to their definitions when the term is italicized, otherwise they can be interpreted just as usual.

literal

program refers to the ATOM program, written in ATOM language, instead of the generated one.

compile-time The period from ATOM receives the input, till the generated program is run.

run-time

terminate dump the data until this point to a persistent medium, and abort the program with no clean up required

3 Variable [var]

1 Variables are any data fields present in the *program*.

2 Programmatic information about a variable is defined by its scope modifier and type name. Those are unordered but must be before the variable identifier. Valid scope modifier are `control` and `global`.

3 There are four kinds of variables: objects, loop control, non-control non-object global variables, and temporary variables.

	scope	type	history persistency
object	global	object	all
loop control	global	any built-in	all
non-object non-control global	global	any	none
temporary	inner loop	any	none
*watch	as parent	any built-in	as parent

4 `control` declares the variable as a loop control, which implies global. It is legal to have both modifier present.

5 `global` declares the variable as a non-control non-object variable if type is not `object`, otherwise an object variable.

6 Every variable has a name served as its identifier.

3.1 Types [var.type]

1 Each variable has a type. For variables other than *object*s, types may be either variant of real or int.

2 No conversion is allowed from float to int or from int to float. Diagnostic is required.

3 Types may associate with qualifiers and/or at most one aggregate modifier, which modifies the type to be a new type. Qualifiers are attributes that provides extra information for ATOM. The qualifier supported is `const`. Aggregate modifiers, when combined with types, give the array form of the type. [*Example:* `real[2]`

is a complete type. — end-example]

4 *Object*s have type of object. No other objects may have type of object.

5 Float type represents a signed non integer number.

6 Int type represents a signed integer number.

3.2 Initialization and Boundary Conditions [var.init]

1 Variables may have boundary conditions that will be verified during the simulation.

2 Boundary conditions may be a compile-time constant or an expression to be calculated at run-time.

3 *Program*s shall define the behavior when a boundary condition is not met.

4 Boundary conditions is checked after any of the values are updated, no later than the value is firstly referenced.

5 Read from an uninitialized temporary variable is undefined.

4 Rule [rule]

1 Rules are routines that are used to generate new data from available data.

2 Rules are universally applied on each object or tuple as requested.

3 Rules may be implicit as equations or explicit as assignments.

4 When requesting object tuple, numbers are used to represent these *run-time* packed objects. Implementation shall return a tuple of objects that is exactly of the length of the greatest number referenced in that rule. [*Note:* The tuple must have a protocol that allow access to its elements with constant time complexity regardless of its length. – end-note]

4.1 Resolution [rule.rsol]

1 Resolution of rules is the process to revealing dependency of rules and generating a specific sequence according to which the rules are solved.

2 Determination of solvability of the system, a specific equation, a specific set of equations, is implementation defined.

5 Input structure [input]

1 A valid *program* has three sections, `object`, `rule` and `init`, exactly one after another. Each section is followed by a pair of curly brackets. Whitespaces are ignored.

2 The `object` section defines watches, delimited by newline. On each line, the input shall have variable declarations prepended by only type name. The scope is handled as per var.3

6 Workflow and Data Generation [datagen]

1 ATOM iterates on loop controls to generate data from initial values. The term *loop* is used to refer to such a structure.

2 In such a loop, ATOM attempts to use *rule*s to generate new data, which are then stored into *variable*s. In other words, *variable*s are then updated by *rule*s.

3 Implementation shall be able to generate either a standalone program or library that can be called by other programs.

4 No external branch shall present in the innermost loop. Data that must be calculated with branches must be ready before the entrance of the innermost loop.

6.1 Mathematical Equivalence [datagen.mathequiv]

1 This section specifies the requirement of the generated solver, specifically by comparing the result to that of a ideal closed-form analytical solution.

2 Real numbers are represented as binary 64-bit float numbers or better, which must at least as absolutely precise as binary floats ($\forall a \in \mathbb{R},\ \exists a_{repr}\ s.t.\ |a_{repr} - a| \leq |a_{float64} - a|$). Representation of NaN or infinities are not required.

3 Integer is stored exactly and possibly as a different type than that for real.

4 Implementation must handle number in $[-2^{31}, 2^{31} - 1]$, with precision no less than 12 decimal significant figures for real, exact for integer.

5 If a mathematical function gets a invalid argument (for example, out of domain), the incident should be reported, and simulation may terminate. [*Note:* Simulation may silently proceed if the program is robust to bad arguments and previous data are kept uncorrupted. – end-note]

6 Infix operators including '+', '-', '*', '/', '^' are accepted as their meaning in mathematical context.

7 An algebraic equation is solved when the absolute difference of lhs and rhs is less than a user-defined tolerance, default to 2^{-27} $\mathbf{max}\{|lhs|, |rhs|, 1\}$. An equation system is solved when the 2-norm of the vector is less than a user-defined tolerance, default to 2^{-21} $\mathbf{max}\{|\text{initial difference}|, 1\}$.

8 Forward derivatives, when there's no future data available, are produced from previous data, by a implementation defined method.

9 Solution at a specific point of a differential equation is valid iff the difference is less than the tolerance of single algebraic equation.

10 Vectors may be defined using array syntax.

11 Addition, subtraction, negation and scalar multiplication of vectors are handled via infix operators as in mathematical contexts. Addition and subtraction with incompatible size is a *compile-time* error.

12 Dot and cross product for vectors are handled via `.mul`.

7 Standard Library [stdlib]

7.1 Support for simulating rule with domain [stdlib.domrule]

[[*Impl*]] Works by having multiple combination generators in the generated program.

7.2 Support for simulating ternary rules [stdlib.condrule]

[[*Impl*]] Works by wrapping the condition as a boundary condition, therefore generates two separate programs and chain them up.

7.3 Support for type nesting [stdlib.nesttype]

[[*Impl*]] Works by generating two separate programs and wrapping one of the nest type as an opaque type (size is known), requires internal support. The outer type will have no access into the inner type.

8 Frontend [front]

1 The frontend is the user interface and the immediate underlying code that handles the input.

[[*Impl*]] The frontend is implemented in Python 3, and the code will remain in Python until it stabilizes.

2 The frontend shall have interface to:

2.1 select a type,

2.2 create a modified type,

2.3 define *object* type from types of *watch*es,

2.4 create a loop control variable of one of the built-in types,

2.5 create a non-object non-control global variable,

2.6 create rule from an equation, and

2.7 initialize or add boundary condition to a variable.

9 Backend [back]

1 Backend refers to the part of ATOM implementation that generates code that is otherwise executable. For this reason, this section defines the interface to ATOM generated code.

2 The generated program shall be completely separate from ATOM support files.

3 A class or other equivalent concept shall be generated and named `object_`, if applicable.

4 All names of *watch*es are preserved.

5 Derivatives of a variable are named in the fashion of `<variable name>_<order>`.