Python to modelica translation guidelines draft

Jáchym Barvínek

Type correspondences A table showing which types should be translated to which.

Python type	Modelica type
`float`	`Real`
`int`	`Integer`

Function parameters and return values For each parameter in the python function, we create a corresponding `input` entry in the modelica code with the same name. Python function parameters are not strictly typed and the type cannot be inferred in general. We will be dealing mainly with arithmetical a numerical calculations and for this reason and sake of simplicity, we well assume that the type is `Real` in cases where it cannot be inferred. We won't translate functions with unknown numbers of parameters now. (The `*` and `**` operators in python.)

Example: test_0008

Return values: Python functions return one value of unspecified type. Thus, the type must be inferred. (Taking the chosen parameter types into account.) We'll assume it's a `Real` as long as the computation is valid. If the return value is a local variable (but nut parameter!), we shall make a corresponding `output` entry in the translation for this variable. If it's an expression or parameter, we make a new variable. This variable is called `return_value` in the tests, another name may be chosen but note that it's necessary to check whether this is not also a name of a local variable or a parameter. In this case, another name must be chosen.

Examples: test_0001, test_0002

Local variables In modelica, there is an ambiguity, where values of local variables can be first assigned. It can be either in the `protected` or the `algorithm` section. I decided to only declare the names in `protected` and assign them in `algorithm`. This makes the code a bit longer, but I thought it could make the translation of the algorithm : bit more straightforward.

Example: test_0006

Side effects In contrast to modelica, functions in python may normally have side effects. A notable example is assignment to arrays. I suggest to solve this by tracking which parameters might be potentially affected by this and adding extra output value to the translated function containing the modified parameter. In the tests, the prefix `modified_` for the variable name, but again it must be checked that the name is not used elsewhere. In the context outside the single function, the translator must also take this into consideration. In the tests, the variable is tracked up to redefinition and then assigned the current value there. If it's not redefined or there is no return value, it's tracked in the whole body of the function.

Examples: test_0014, test_0015

Important note. The arithmetic-assignment operators in python cause side-effects. But assignment to an arithmetic expression does not. For example: If `a` is an array and a parameter of a function in Python which was not redefined, the code `a = a + x` does not change it's original value (but redefines values), but the code `a += x` does change original and doesn't redefine locally!

Example: test_0018

Array size inference When the translator recognizes a variable as array (or simply assumes that it's one), it needs to decide it't dimensions and sizes in each dimension. In Python, if a operation works on an array of n dimensions, it may also work on array of any higher dimension. But in basic modelica array semantics, it's not easily possible to handle arrays with dynamic number of dimensions. Thus, the translation assumes the lowest possible dimension that makes sense according to arithmetical operations found in the algorithm (but at least one, zero dimension only in explicit cases like `numpy.array([])` or `numpy.empty((0,))`.) If the arithmetical operations do not provide any evidence, that the variable (or parameter) is an array, it should be assumed to be a scalar number. If sizes of the array in some dimensions cannot be inferred from arithmetical operations, they shall be unknown (i.e. `[:]` in Modelica). We do not know any good way to translate a function which takes as parameter an array of number of dimensions which cannot be inferred (or lower bounded). Now, the translator should issue an error when it detects such situation.

Examples: test_0007, test_0013.

One important things to note is that some arithmetical operations on arrays cannot be translated in a straightforward way, for example, modelica does not support addition of arrays of different dimensions but in python it's possible under certain conditions. The translator must take care of this. For example: `numpy.array([[1,2],[3,4]]) + numpy.array([7,5])` results to `array([[8,7],[10,9]])`. In the translation, I solve it with a for loop.

Examples: test_0017, test_0019

Note also, that in python, although `numpy` arrays have fixed size, the variable that contains them

Anonymous functions Anonymous (`lambda`) functions are taken out of the surrounding function and translated as normal function with a generated name.

Example: test_0020.

Lists (MetaModelica) Lists in python can be heterogeneous, i.e. a single list can contain elements of various types. In practice, and especially regarding numerical computations on which we focus, lists are most often homogeneous, though. Translating heterogeneous lists would be possible in theory, but it would be very complicated and probably resulted in swollen code. Let's assume, that lists in code we translate are homogeneous. If the type of a list cannot be inferred from the function code, we shall assume that it's float. If there is a certainty, that the list is heterogeneous, the translator may try another strategy (such as converting the list to tuple, but that may cause other problems) otherwise it should issue an error.

TODO multidimensional lists, list index assignment, slicing. Problem: Some basic constructs cannot be translated using builtin MetaModelica functions on lists.

Tuples (MetaModelica) Tuples in python are simillar to tuples in MetaModelica, but it might be difficult to determine their type, even the length may be uncertain. If the python code provides clues about the size and types, it's simply translated to corresponding tuple in MetaModelica.

It may be sometimes impossible to determine, if a variable is a `list` / `tuple` or `numpy.array`. In that case, it should be assumed to be `numpy.array`.

Options (MetaModelica) In python, each function has a return value, if the return value is not specified, it's `None`. If no `return` statement is in a function, only side-effects are returned. (If there are no side effects and no return value, the function does nothing and can be replaced by constant `None`). This suggest itself to be translate a function which may return either a value or `None` using MetaModelica options.

But note, that the resulting type in Modelica will be `Option<Real>` for example, but in python it's simply either `float` or `NoneType`. The translator must take this in consideration when later dealing with the returned value.

Typical situation is when a condition in `if` statement asks if certain variable is `None`. In that case, it's safe to assume, that the variable in question should have type `Option<...>` when translated. This makes translation of `if` statements seemingly more complicated, but notice that such statements would not be translatable anyways, since variables in Modelica are strongly typed. Thus, this is an unambiguous extension to the `if` statement translation.

Built-in functions This table shows correspondence between built-in functions and basic language constructs used in the tests.

Python	Modelica	Notes
`range(a,b,c)`	`(a:c:b+1)`	
`range(a,b)`	`(a:b+1)`	
`range(a)`	`(0:a+1)`	
`numpy.arange`		same as `range`
`for x in z: ---`	`for x in z loop --- end for;`	`x`, and the block `---` must be translated. `z` is a `range`.

`for x in lst: ---`	`for local_variablei in`
	`(1:listLength(lst)) z loop`
	`x := listGet(lst, local_variablei);`
	`--- end for;`
		`l` is a `list`.

`while b: ---`	`while b loop --- end while;`	`b` and the block `---` must be translated

`if b1: ---`
`elseif b2: ---`
`else: ---`
	`if b1 then ---`
	`elseif b2 then ---`
	`else --- end if;`
		`b1`, `b2` and the corresponding blocks `---` must be translated. See also paragraph about MetaModelica options.

`numpy.ones((a,b,c,...))`	`ones(a,b,c,...)`	
`numpy.ones([a,b,c,...])`	`ones(a,b,c,...)`	Alternative syntax.
`numpy.zeros((a,b,c,...))`	`zeros(a,b,c,...)`	
`numpy.zeros([a,b,c,...])`	`zeros(a,b,c,...)`	Alternative syntax.
`numpy.empty((a,b,c,...))`		Array of given dimensions is only declared!
`numpy.empty([a,b,c,...])`		Alternative syntax.
`numpy.dot(a,b)`	`a*b`	Matrix product.
`a.dot(b)`	`a*b`	Alternative syntax.
`numpy.array`		array constructor
`numpy.concatenate((a,b,...), z)`	`cat(z,a,b,...)`	parameter `z` may be omitted in python or given as `axis=z`
`numpy.concatenate([a,b,...], z)`	`cat(z,a,b,...)`	Alternative syntax.
`m.fill(v)`	`m := fill(v, a,b,...)`	Where `a,b,...` are dimensions of array `m`.
`numpy.sum(x)`	`sum(x)`	
`numpy.product(x)`	`product(x)`	
`[x][y]`	`[x,y]`	Indexing (general)
`[x,y]`	`[x,y]`	Indexing (`numpy` arrays only)
`:a`	`1:a`	Array slice indexing
`a:`	`(a+1):end`	Array slice indexing
`a:b`	`(a+1):b`	Array slice indexing
`numpy.identity(n)`	`identity(n)`	
`numpy.linspace(a,b,c)`	`linspace(a,b,c)`	
`m.shape`	`size(m)`	
`l.append(e)`	`l := listAppend(l, {e})`	`l` is a `list`.
`l1 + l2`	`listAppend(l1, l2)`	`l1`,`l2` are `lists`. `+=` Has side effect.
`l1.extend(l2)`	`l1 := listAppend(l1, l2)`	Has side effect.
`del lst[i]`	`listDelete(lst, i)`	Has side effect.
`len(lst)`	`listLength(lst)`	Assuming `lst` is a `list`, not a `numpy.array`!
`lst[i]`	`listGet(lst, i-1)`	Also assuming `i >= 0`.
`lst[-i]`	`listGet(lst, listLength(lst)-i+1)`	
`x in lst`	`listMember(x, lst)`	Different semantics of `in` than in `for` loop statement.
`math.asin`	`asin`	