Fudomo Tutorial (version 0.1)

Outline

Introduction

Fudomo

- Fudomo is ...
  - A tool - package for ATOM editor that supports example-driven modeling as well as model to text transformations
  - A declarative language for defining model to text transformations
  - A hybrid approach to model transformations based on combining model and code
- What FuDOMo stands for:
  - **Fu**nctional **D**ecomposition over **O**bject **Mo**dels

Introduction

Installation Instructions

- Download and install ATOM text editor from atom.io
- Install *language-fudomo* package:
  - Open Atom Preferences
  - Click on Install
  - Enter package name *language-fudomo* in text box and then install
  - Install any needed dependencies upon restart (answer Yes in all dialogs)
- Install *linter-js-yaml* package using a similar procedure

Introduction

Installation Instructions (2)

- Fudomo allows you to implement transformations in either Javascript or Python
- It is recommended that you install IDE support for the language of your choice as well, e.g., package *ide-python* for Python and *ide-typescript* for Javascript support

Introduction

Basic Workflow

- The simplest way to use Fudomo goes through the following three steps:
  - write object model
  - write transformation
  - execute transformation
- We will illustrate this basic workflow on a very simple example

Introduction

First Transformation: Hello World!

- Generally transformations we write in Fudomo take an input (object) model and produce an output
- Our first "transformation" is trivial: it simply outputs "Hello World!"
- Let us call the transformation function "helloWorld"

Introduction

Step 1: Write Object Model

- An object model comprises typed objects
- Our first transformation does not need any input
- Since Fudomo needs an input object model in all cases, we supply an empty file named "helloworld.yaml"
- The corresponding object model is not really empty
  - it is assumed to contain a single object of type *Root*
  - Root is a singleton (implicit) type

Introduction

Step 2: Write Transformation

- Fudomo is a *hybrid transformation approach* in the sense that the transformation is represented by a model as well as code
    - The model - called *Fudomo model* - describes the decomposition of the transformation function into simpler functions
    - The (source) code describes how the results of these simpler functions are combined to produce the result of the more complicated functions
        - Currently (as of November 2019) Javascript and Python are supported for writing this code
- The transformation is thus represented by two files:
    - File that contains the Fudomo model: extension ".fudomo"
    - File that contains the code: extension "".js" or ".py"

Introduction

Fudomo Model: Typed Functions

- The Fudomo model deals with *typed functions*: these are functions defined in the context of a type occurring in the object model
- Let us call our transformation function helloWorld.
- helloWorld is also a typed function; its context is the (implicit) *Root* type
- Hence we will denote our first transformation by the typed function

```
Root.helloWorld
```

Introduction

Fudomo Model: Decompositions

- The Fudomo model consists of a sequence of decompositions
- Each decomposition describes how to decompose a typed function (left-hand side) into simpler (typed) functions (on the right-hand side)
- Since the helloWorld transformation function produces a constant result (i.e., indepedent f the input), it does not need to be decomposed further
- This is expressed by the Fudomo model having a single decomposition whose right-hand side is empty

```
# file: helloworld.fudomo
Root.helloWorld:
```

Introduction

Generating Function Skeletons

- From the decompositions in the Fudomo-model we can generate function skeletons in one of the supported languages
  - To generate the function skeletons, right-click on the fudomo-file and select "Generate Function Skeletons > Javascript or Python"
- Here are the generated function skeletons (only one in this case) in Javascript for the helloworld transformation function

```
# file: helloworld.js
module.exports = {
  /**
   * Root.helloWorld
   */
  Root_helloWorld: function() {
    throw new Error('function Root_helloWorld() not yet implemented');// TO DO
  },
};
```

Introduction

Implementing the Decomposition Function

- We call the function *Root_helloWorld* the **decomposition function** for transformation function *Root.helloWorld*
- Its purpose is to compute the result of the transformation function based on the results of the functions it is decomposed into
  - the results of the constituent functions are passed as parameters
  - in this case there are no parameters since the decomposition is empty
- Note that the decomposition function is *not* a typed function but rather a *pure function* in a sense that its value is determined by the values of its parameters

Introduction

Implementing the Decomposition Function (2)

- We implement the decomposition function (in Javascript) by simply returning "Hello world!"

```
# file: helloworld_functions.js
module.exports = {
  /**
   * Root.f
   */
  Root_helloWorld: function() {
    return "Hello World!";
  },

};
```

Introduction

Putting the Pieces Together

- We need to tell the Fudomo-tool where the different pieces are that make up the transformation function, namely:
  - the location of the Fudomo-model
  - the location of the source code implementing the decomposition functions
  - the location of the input model
  - the location of the output file
- We specify this information in configuration file *helloworld.config*

Introduction

Putting the Pieces Together (2)

- Here are the contents of the config-file for our transformation
- We assume that all files are located in the same directory
- We recall that the input file *helloworld.yaml* is empty

```
# file helloworld.config
decomposition: helloworld.fudomo
functions: helloworld_functions.js
data: helloworld.yaml
output: helloworld.output
```

- To execute the transformation, right-click on the config file and select "Run Fudomo Transformation"
- This will generate the single line output "Hello World!" in the output file

Notation for Object-Modeling

Towards OYAML

- Fudomo is an example-driven modeling approach
- The basic workflow starts with creating a concrete example, which we call **object model**
- We need a notation for object models
- We present a new notation for object models, named OYAML, that is based on YAML
- So let us start by introducing YAML first

Notation for Object-Modeling

What is YAML

- YAML stands for: YAML Ain't Markup Language
- What it is: YAML is a human friendly data serialization language.
- Broadly used for programming related tasks:
  - configuration files
  - internet messaging
  - object persistence
- YAML 1.2: http://yaml.org/spec/1.2/spec.html

Notation for Object-Modeling

What is YAML (2)

- YAML organizes data into different levels
- YAML marks levels of data by indentation:
  - Data at the same level are aligned to the left with the same indentation
  - Use spaces instead of tabs for indentation
  - You can use any number of spaces to indicate one level of indentation
- YAML supports the following data structures:
  - Scalars
  - Mappings
  - Sequences

Notation for Object-Modeling

YAML Scalars

- Scalar values (scalars in short) are the most basic and indivisible data
- Scalar types and example scalar values:
  - String: Paul, Cat
  - Boolean: True, False
  - Integer: 5, 8
  - Floating Point: 3.14159, 314159e-05

Notation for Object-Modeling

YAML Mappings

- A mapping is a set of key-value pairs
  - use a colon and space (": ") to separate the value from the key
  - each key-value pair starts on its own line
- Example:

```
lastName: Smith
firstName: Paul
age: 18
isMarried: False
```

- Translate into JavaScript dictionary

```
{ lastName: 'Smith',
  firstName: 'Paul',
  age: 18,
  isMarried: false }
```

Notation for Object-Modeling

YAML Sequences

- A sequence is a list of data
  - use a dash and a space ("-␣") to indicate each entry/element in the sequence
- Example:

```
- Cat
- Dog
- Goldfish
```

- Translate into JavaScript array

```
[ 'Cat', 'Dog', 'Goldfish']
```

Notation for Object-Modeling

YAML Complex Structures

- Sequences and mappings can be nested
- Example (using "#" to start single-line comments):

```
languages:# first dictionary entry
# value is a sequence - note the indentation
  - YAML
  - Ruby
  - Perl
  - Python
websites:# # second dictionary entry
  # value is a dictionary - note the indentation
  YAML: yaml.org
  Ruby: ruby-lang.org
  Python: python.org
  Perl: use.perl.org
```

Notation for Object-Modeling

YAML Complex Structures (2)

- Translate into JavaScript:

```
{ languages: [ YAML ', 'Ruby', 'Perl', 'Python' ],
  websites:
   { YAML: 'yaml.org',
     Ruby: 'ruby-lang.org',
     Python: 'python.org',
     Perl: 'use.perl.org' } }
```

Notation for Object-Modeling

From YAML to OYAML

- OYAML is a sub-language of YAML for object (example) modeling
- Top level data structure is a sequence
- Each element of the top level sequence represents an *object*
- An object is represented by a singleton mapping:
  - key:␣value
  - key indicates the *type* (and optionally the *id*) of the object
  - when the id is present, type and id are separated by a space, as in "Type␣id"
    - Note: types must start with upper case letter.
  - value gives the content of the object

Notation for Object-Modeling

Two Kinds of Objects in OYAML

- *Simple object*: the value of the object is a scalar
- *Composite object*: the value of the object is captured by a sequence. Elements of the sequence can be either:
  - an *attribute*, written as "attributeName:␣attributeValue", where attributeValue is a scalar value
  - a *reference*, written as "referenceName␣>:␣referenceValue", where referenceValue is a comma-separated list of object ids
    - Note: attribute and reference names must start with a lower case letter
  - a *contained object*

Notation for Object-Modeling

OYAML Example: familySmith.yaml

```
# simple object followed by composite object
- Address add0001: 1 route de Luxembourg, L-1234 Belval
- Family smith:
  - familyName: Smith # attribute
  - address >: add0001 # reference
  - Member jim: # contained objects start here
    - name: Jim
    - age: 45
  - Member cindy:
    - name: Cindy
  - Member brandy:
    - name: Brandy
    - age: 18
  - Member toby:
    - name: Toby
    - age: 12
```

Notation for Object-Modeling

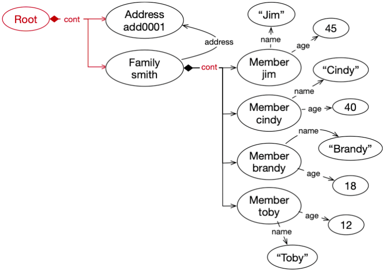
OYAML Containment Tree

- All objects in an OYAML model are either a top-level object, or an object contained in a top-level object, directly or indirectly.
- On the following page we show the containment tree for the previous example

Notation for Object-Modeling

OYAML Containment Tree for familySmith.yaml

- Note the inclusion of an implicit *Root*-node and containment being represented by implicit reference "cont"



A Second Transformation

Definition

- We define the second transformation with the help of an example
- From a sequence of families, e.g.:

```
- Family:
  - familyName: Smith
- Family:
  - familyName: March
- Family:
  - familyName: Rafton
```

- we want to generate a comma-separated list of family names. In this case:

```
Smith, March, Rafton
```

A Second Transformation

Functional Decomposition

- Call the transformation function that produces the desired list of names `family2names`
- We thus need to decompose typed function `Root.family2names`
- For this we need to dig a bit deeper into typed functions and decompositions

A Second Transformation

Typed functions Revisited

- Let T.f be a typed function for a type T
- To compute the value of a typed function, we need to fix an object model m and an object x in m of type T
- We denote the value of f for m and x by: x.f(m)

A Second Transformation

Decomposition: Syntax

- A decomposition in a Fudomo model consists of a left-hand side and a right-hand side separated by a colon
- The left-hand side contains the typed function (call it $T.f$) to be decomposed
- The right-hand side contains a comma separated list of typed functions sharing the same context type as f
- Thus the general form of a decomposition is: $T.f\colon f_1, \ldots f_k$
- Note that we omit the context type on the right-hand side

A Second Transformation

Decomposition: Semantics

- The meaning of a decomposition $T.f\colon f_1, \dots f_k$ is that $T.f$ is *determined* by the values of $f_1, \dots f_k$
- More formally, it means that for any object model m and object a in m of type T, $a.f(m)$ can be computed from $a.f_1(m), \dots, a.f_k(m)$
- We call the function that performs this computation the *decomposition function* for f

A Second Transformation

Decomposing family2names

- So how do we decompose the family2names transformation?
- Note that the value of `Root.family2names` is determined by the names of all families contained in the Root object

\lstinline{A.r -> B.g}

- whose value is $b1.g, b2.g, \ldots, bk.g$

A Second Transformation

Back to family2names

- Recall that the value of `Root.family2names` is determined by the values of the *familyName* attribute for each family contained in the root
- In other words: `Root.family2names` is determined by forward function
  - `Root.(cont -> Family.familyName)`
- We write this in the Fudomo file as follows:
  - `Root.familyNames: cont $->$ Family.familyName`
- Note that we write `cont -> Family.familyName` in the RHS instead of "Root.(cont $->$ Family.name)" because the context Root is clear from the LHS
- How about the Family.familyName typed function? Does it need to be decomposed further?
  - No, because familyName is an attribute of Family so this "function" can be directly "looked up" in the object model.

A Second Transformation

family2names: Solution

- See below the fudomo model (left) and the code that implements the decomposition function in Javascript (most of which is generated)

```
# File: family2names.fudomo
Root.familyNames: cont -> Family.familyName
```

```
# File: family2names.js
module.exports = {
  /**
   * Root.familyNames:
   * @param cont_Family_familyName {Array} The sequence of
   * "familyName" values of Family objects
   * contained in this Root
   */
  Root_familyNames: function(cont_Family_familyName) {
    # next line added manually, rest generated
    return cont$_$Family$_$familyName.join(', ') + '\n;}}

  },
};
```

A Third Transformation

Definition

- Next example: families with contained members

```
#INPUT
- Family:
  - lastName: March
  - Member:
     - firstName: Jim
     - sex: male
  - Member:
     - firstName: Cindy
     - sex: female
  - Member:
     - firstName: Brandon
     - sex: male
  - Member:
     - firstName: Brandy
     - sex: female
```

```
# OUTPUT
Mr. Jim March
Mrs Cindy March
Mr Brandon March
Mrs Brandy March
```

A Third Transformation

Decomposing the Transformation Function

- Denote the transformation function by `Root.persons`
- Let typed function `Family.persons` compute the list of persons (as shown on the previous slide) for a given family
- We note that `Root.persons` is determined by the values of Family.persons for all the families (contained in the root)
- We can thus decompose Root.persons with a forward function as follows:

```
Root.persons: cont -> Family.persons
```

- Note again that we can omit the context type of the forward function because it is the same as the context type of the `persons` function

A Third Transformation

Decomposing the Transformation Function (2)

- Next we note that `Family.persons` is determined by the full names (including "Mr" or "Mrs") of the members contained in this family
  - Call the typed function that returns the full name of a family member: `Member.fullName`
- Thus we get the next decomposition, again using a forward function:

```
Family.persons: cont -> Member.fullName
```

A Third Transformation

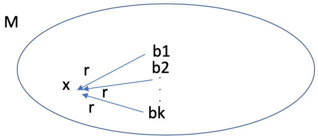
Decomposing the Transformation Function (3)

- The full name of a Member depends on its sex, its first name and the last name of the family it is contained in
- Note that for the last name we need to follow the "cont" reference backwards!!
- We therefore call the corresponding function a *reverse function*

A Third Transformation

Reverse Functions

- Fix types A and B and reference r
- Denote by $A.(r < -B.g)$ the typed function for A defined as follows:
  - $A.(r < -B.g)(M, x) = \{g(M, b1), \ldots, g(M, bk)\}$
- where b1,...,bk are the objects of type B that refer to x via reference r



Informally: start from x of type A, follow backwards reference r and evaluate g on each of the visited objects of type B

- We call $A.(r < -B.g)$ the **reverse function** (for reference r and type A)

A Third Transformation

Decomposing the Transformation Function (4)

- We are now ready to write the next decomposition:

```
Member.fullName: sex, firstName, cont <- Family.lastName
```

- Note that sex, firstName and lastName are attributes that do not need to be decomposed further (since they can be directly looked up in the example)

A Third Transformation

Implementing the Transformation Function

- We are now ready for the implementation
- This time we do it in Python!
- Before we can do this we must generate the function skeletons:
  - Right-click on Fudomo file containing the above decompositions and choose "Generate Function Skeletons > Python 3"
- Take a look at the function skeletons
- Now implement the functions

A Third Transformation

Solution

```
# File: family2persons.fudomo
Root.familyNames: cont -> Family.familyName
```

```python
# File: family2persons.py
def Root_persons(cont_Family_persons):
    """
    :param cont_Family_persons: The sequence of
    "persons" values of Family objects contained in
    this Root
    :type  cont_Family_persons: Array
    """
    return ', '.join(cont_Family_persons)

def Family_persons(cont_Member_fullName):
    """
    :param cont_Member_fullName: The sequence of
    "fullName" values of Member objects contained
    in this Family
    :type  cont_Member_fullName: Array
    """
    return ', '.join(cont_Member_fullName)
```

A Third Transformation

Solution (2)

```python
# File: family2persons.py - continued

def Member_fullName(sex, firstName, _cont_Family_lastName):
    """
    :param sex: The "sex" of this Member
    :param firstName: The "firstName" of this Member
    :param _cont_Family_lastName: The set of "lastName"
    values of Family objects that contain this Member
    :type _cont_Family_lastName: Set
    """
    familyName = next(iter(_cont_Family_lastName))
    prefix = ''
    if sex == 'male':
        prefix = 'Mr '
    else:
        prefix = 'Mrs '
    return prefix + firstName + ' ' + familyName
```

A Third Transformation

Validating Transformation and Object Models

- A transformation is *valid* if it uses types, attributes and references that are contained in the metamodel of the object model
- To validate transformation:
  - First infer metamodel from object model
    - Right-click on oyaml file and select " Infer Metamodel"
  - Now validate transformation with respect to metamodel by right-clicking on config-file and select "Validate Fudomo transformation"

A Third Transformation

Validating Transformation and Object Models (2)

- We can also validate other object models wrt inferred metamodel
  - Make sure "metamodel" field is set
  - Set the data field to the other object model
  - Right-click on config-file and select "Validate data model"
- Note: When the metamodel field is not set, executing "Validate data model" will check basic OYAML syntax of the object model (independent of metamodel)