**Documentation about realized software for sending computation**

**Introduction**

Some of the inputs of this work have been the work of Marco Rodriguez about the RDF virtual machine, the Haskell state monads and various papers about out of order execution like…[cite]

The objective was to realize and to describe semantically a scenario in which computation can be represented in a machine understandable way and sent to another execution environment where, using the semantic, it can be reconstructed and computed. The computations target are not low level machine level instructions like in Rodriguez work, nor simply functions, but a particular kind of function that can be executed or put in chain with other functions of the same type like it happens with monadic values that can be put in chain with monadic operators like >>= ( bind). Another important objective of the semantic representation of computation in this work was to realize a system in which computations originated by a certain HW/SW combination, can be executed independently from the executor operating system or hardware architecture. One of the possible execution environments is provided by a big number of specialized execution units and some coordinator unit which decomposes and assigns computations to the executors and then routs back the results to the requestor. Starting from a code written in a certain language, particular functions or chain or functions can be detected in a preprocessing phase and some scheduler can decide, depending from context parameters like memory/processor usage or battery level, if computing locally the functional chains pr delegate ti a remote unit. The execution of the rest of the code can continue until the results of he functional chain are effectively needed by some function call; in this case if the results have still not been received from the remote executors, the program blocks waiting for them not . Another important feature to be taken in consideration is the level of granularity of the functions we want to represent in a semantic way. At machine instruction level there is a big verbosity, but the representation of the instruction of an ISA allows all possible functionality, so it is general purpose. At higher expressivity the number f the functions to be represented functionally can be not upper-bounded in general, but subsets of the possible functions can be detected to describe specific domains like image processing or compressing or others. High expressive semantically characterized functions or functional chains can be used as building blocks of efficient applications with specific execution units to execute with high performance the recurrent functionalities. This last and some of the previously described objectives can be planned and considered in more concrete terms only after the availability of a demonstrative implementation with the purpose of demonstrating that the approach is at least realizable.

The current prototype for function semantic representation and computation is not a complete work usable to create applications based on it, but represents a proof of concept demonstrating various results that is possible to achieve by working on the way of representing pieces of computation and demanding their execution to remote units. The code needs modularization and polishing, at moment the attention has been focused only at correctness of results, and proper semantic representation

To store the functional chains in a semantic format and retrieve them the smart M3 platform of Nokia has been used. The platform allows for the agents to subscribe to specific information and this characteristic has been used where possible to have a more automatic execution and result retrieval.

**The execution scenario**

In figure 1 we can see how a typical computation is currently executed:

1. The execution manager subscribes to the semantic platform in order to be notified of the new computations.
2. A caller program constructs functional chain made up of one or more functional blocks and send it To the semantic information
3. The execution manager is notified and retrieves initial information about the functional chain to compute
4. The execution manager starts a chain executor in a new thread to perform the computation, and is again ready to receive new functional chains
5. The chain executor starts one function executor for each function in the functional chain.
6. Calculated results are written I n the shared space
7. Final results are notified to the caller

The execution flow can be also more complicated than that represented, the functions executors, in example, can be dependent one from the result of the others and so additional subscriptions- notifications are needed to bring the whole computation to end. This execution flow maximizes parallelism, in fact, when a function executor is ready to run, it runs like in a data-flow architecture. Function executors are now execution threads, but in principle they can correspond also to different HW that can be specialized the execution of the function for which it was called.

**Closures port and parameters**

We called “Closure” a concept corresponding to a functionality, it inputs and its outputs. A closure can be drawn as a box with slots called ports. The port is the instrument by which values are associated to the function as input or output. The Ports are associated to a parameter that is the container of value or of a reference to it. The ports are also provided by a name whose function is to allow the correct reconstruction of the closure when it is queried by the SIB. A closure with inputs all provided by a Value is ready to run in the sense that the functionality corresponding to the closure itself can be executed on the inputs to obtain the outputs. The first closure realized performed the functionality of summing two integers. So it was provided by two input ports (In1 and In2) and one output port ( Out). The parameter without a value are conventionally called, the value can be calculated by other closures in the same chain of functions or by external processes, whoever, when writing the value on the triple store has also to take care of deleting the invalidity statement and put a validity one for the parameter. In general a closure sent to the RDF store with at least one invalid parameter, correspond to a waiting thread, when al the inputs will be valid the thread will run its functionality and will write the values on the corresponding output parameter. An invalid parameter can be connected to the input port of a closure eand also to the outputport of another. In thisway a connection between the two closure is realized and when the closure with the parameter as output will write it, then the other one with the invalid input will be possibly ready to run and bring on the computation.

Each parameter value has to be transformed into a String to be put in the RDF store, so the elementary datatypes that can be given to a closure need a toString() method and probably need a constructor that calculates the original value starting from the String (i.e. in the reconstruction phase). Since this kind of extension (i.e. supporting various datatypes) seemed to be trivial and time consuming, the work done has been focused on Integers initially. The parameters may not only refer to a single value, but also to many values of the same type ( Vector-Parameters ) or to functions (Functional-Parameters). The caller construct a closure by calling its constructor and telling the value of the parameters if known or a reference to them if it is necessary some closure connection.

A generic Closure constructor has not been optimized, so for some of the realized functionalities it will be possible to notice too many parameters in the constructors, but this was done to have a regular software structure that can be modified in every moment. Before explaining how the constructor is done is necessary to introduce the different parameter types. A parameter may not only correspond to a single value, but also to an array of values or to a function. A Vector-parameter is an entity with at least a URI, a type and a dimension to which are attached single parameters. A function-parameter is an entity with a URI to which is attached a functional chain containing one or more closures. The current closure constructor takes a Vector of Parameters for each parameter type and it can take also values instead of parameter, if it is not necessary to specify the URI of the parameter itself. If for a certain index both the value and the parameter will be null the constructor creates an empty parameter (invalid parameter or parameter without a value). The port of a closure are considered named and ordered, currently each closure have ports statically defined also if it is possible to think to closures with a dynamic number of ports. An example of closure with a dynamic number of port could be the one whose aim is to take a vector and give the parameters it contains(dynamic number of outputs).

The closure constructors should be callable by a sort of preprocessing task in order o create some possibly complex chain. By the use of an hashtable for parameters connection it has been possible to perform this task in the FunctionalChain Class. It is possible to add functions to a functional chain by providing for the parameters values or references (“in the form of literals”). Through the use of the hashTables the literals are mapped to the URI of the parameters and so if the same literal is used to initialize two closure parameters, since the URI of the parameters will be the same, also if starting from two different objects, when writing on the RDF the Parameter will be the same and the two closures will result connected. All this procedure can be probably simplified or optimized, but as said this is true for most of the code in the current version.

**Examples**

**Examples**

By going through two examples it should be possible to understand state and potentiality of the current implementation

String[] inputs = **new** String[2];

String[] inputsRef = **new** String[2];

String[] outputsRef = **new** String[1];

inputs = **new** String[2];

Vector<String[]> VecInputs = **new** Vector<String[]>();

Vector<String> InReferences = **new** Vector<String>();

String[] inValues = {"1", "2", "3"};

VecInputs.add(inValues);

outputsRef = **new** String[1];

outputsRef[0] = "a";

addVecFunctionCall(OntologyVocabulary.*AddVectorIntClosure*, VecInputs,InReferences, **null**, **null**, **null**, outputsRef);

inputs[0] = **null**;

inputs[1] = "2";

inputsRef[0] = "a";

inputsRef[1] = **null**;

outputsRef = **new** String[1];

outputsRef[0] = "b";

addFunctionCall(OntologyVocabulary.*AddIntClosure*, inputs,inputsRef, outputsRef );

Vector<Vector<String>>triples= serializeTriples();

The written code correspond to the creation of a closure chain made up of two closure: one adds all the element of the input vector and one simply sum two integers. The output of the first one is called “a” that is the same reference given for the first input port of the second closure, so there will be a single parameter “a” initially without value that will be written by the first closure by allowing the second one to execute. In figure 2 it is represented schematically what happens in the RDF store: in blu the single parameters, in orange the Vector-parameters with dimension inside in red what is calculated at execution time.

In the execution phase a graph made up of several triples representing in detail the computation is received by the SIB and the execution manager, that recognizes the closure chains, starts a chain executor. The chain executor discovers that there are two functions inside the chain and starts two function executors. One of the two begins to execute soon, the other one has an invalid input parameter and waits for it to be written by some process. At a certain point the first executor calculates the results (i.e. 6) and writes it back in the SIB by also updating the validity information of the parameter; now the second function executor discovers the availability of the new input, it queries for it and finally executes calculating the value and writing it back into the SIB. The caller, subscribed to the value of the parameter named “b”, is now able to have to know it. A similar execution flow happens in case of more complicated closure chains like the one represented in Fig.3: also the vector-parameter can be invalid and the written by some function executor, moreover is possible to notice that this is true also fir the single parameters inside the contentment of the vector parameter. The routing of vector and single parameter is very flexible also if , with the current implementation the declarative code to specify such parameter routing is a bit verbose

A different execution flow is the one that happens when function-parameters are used. The function parameters are used in the case of HOF (Higher order functions) which have the good property of being very expressive. The work done supports function- parameters and HOF only partially with the purpose of being a proof of concept of the reliability of HOF with the proposed framework and of their ability to be represented by RDF triples.

The following code the starting situation: what is sent to the SIB :

FunctionalChain fc = **new** FunctionalChain();

String[] inputs;

String[] inputsRef;

Vector<String[]> VecInputs = **new** Vector<String[]>();

Vector<String> InReferences;

String[] outputsRef;

inputs = **new** String[2];

inputsRef = **new** String[2];

outputsRef = **new** String[1];

inputs[0] = "3";

inputs[1] = **null**;

inputsRef[0] = **null**;

inputsRef[1] = "in";

outputsRef[0] = "f\_x";

fc.addFunctionCall(OntologyVocabulary.*AddIntClosure*, inputs,inputsRef, outputsRef );

inputs = **new** String[2];

inputsRef = **new** String[2];

outputsRef = **new** String[1];

inputs[0] = **null**;

inputs[1] = "4";

inputsRef[0] = "f\_x";

inputsRef[1] = **null**;

outputsRef[0] = "f\_y";

fc.addFunctionCall(OntologyVocabulary.*SubIntClosure*, inputs,inputsRef, outputsRef );

FunctionalParameter fp = **new** FunctionalParameter();

fp.setContent(fc);

FunctionalParameter[] forInitialize = **new** FunctionalParameter[1];

forInitialize[0] = fp;

String[] inValues = {"1", "2", "3"};

VecInputs.add(inValues);

VectorParameter outParameter = **new** VectorParameter();

outParameter.setRandomURI();

//connectionTable.add

outParameter.setValid(**false**);

VectorParameter[] mapoutputs = **new** VectorParameter[1];

mapoutputs[0] = outParameter;

addFunctionalFunctionCall(OntologyVocabulary.*MapClosure*, VecInputs, **null**, mapoutputs,**null**, **null** ,**null**,forInitialize , **null**);

Vector<SingleParameter> outs = AtomicFunctions.lastElement().getVectorialOutputPorts()[0].getSignal().getContent();

**for** (**int** i = 0; i < outs.size();i++)

{

connectionTable.put(outs.elementAt(i).getURI(), ("mapout\_" + i) );

}

In figure 4 is represented the situation at information level: what is red or surrounded by red is not initially written into the SIB and in particular is written by the function executor of the map method and its sub threads. The green parameter and ports correspond to the flow of information of functional parameters. The functional chain input of the Map method has one input and one output so these parameters are not needed in the functional parameters since they lack for the intrinsic nature of the map method. The map operation has as input an array and a function and gives as output an array containing the result of the application of the input function to each of the value inside the input vector. To represent and do this in our framework there are various possible way with pros and cons, but some consideration can be done. Each HOF has as input one or more functions and does something with them, the input functions are so used in two different ways during the process: at the beginning they are simply an input without any objective, and later their functionality has to be applied in some way. We call these two phases representational phase and execution phase after expansion. In the case of the map the execution correspond to the application of function to the right input and targeted to the right output. This objective can be done for example by repeating the functionality embedded in the functional parameter for all the input and putting the result in the right slot of the output vector. It is also possible to run in parallel the functionality on the various inputs by exploiting the parallelism implicit in the map operation. The solution that has been implemented currently is different to exploit parallelism and to show that not only closure can be reconstructed by a series of query, but also user defined chain of function which becomes de-facto first class entities in our framework. When the function executor understand that it has to perform a map operation the input functional chain is reconstructed and then replicated as many time as the dimension of the input vector. At this point for each replica the input and the output references are correctly bound (red arrows) and the obtained executable closure chains are simply sent again to the SIB firing three times the execution manager. Performances are not obviously the objective of this implementation because, as previously explained, the final scenario is quite different from the current execution environment and now the most important task is to understand what is possible to do with computation semantically represented to proceed in another moment to the refinement.

**Limitations and possible extensions**

Here some of the current intervention current implementation which should be done

* Vectors are not managed recursively : we have only vectors of single parameters and not vectors of Vectors and vectors o functions. A complete management o vectorial parameters can be very helpul or the expressivity of the system (not trivial)
* The functional input of the map closure now works only with ready inputs and only single parameters are allowed inside the input functional chain. Simple modification to have a more complete map method can be useful.
* Adding to the system parameter with complex values like user defined data-type with a toString method (not so much difficult, perhaps long)
* Adding to the system the management of cost functions and annotations related with performance and resource consumption in order to have different execution strategies depending from context server side or execution planning at client side
* Adding to the system particular parameters with a reference to the value (e.g. images or videos) This could be useful for many use-cases in which resources are not on the device and the download of the data is avoided: is downloaded only the result (seems not trivial) but very important and not impossible. In the case of very big closures like compression algorithms or images processing it could be a very simple way to move computation away and leave the device resources free
* Supporting lot of functions to have the possibility to test benchmarks.
* As exercise is possible to create the equivalent of the bind for the state monad and chain some function through it. The bind is an Higher order function with two functions as input and a function as output.
* Supporting closures with a dynamic number of ports
* Optimize code to be more efficient

**How to run tests.**

There is some test of the platform in the functional chain class. To run them is necessary to

* Install and run sib daemon and sib-tcp if we want to run locally.
* Be sure that the strings in the SibConstants class point to the wanted SIB implementation
* Choose one of the run method in the Functional chain Class and calling it from the main.
* Similar programs performing different functionalities can be realized looking at the existing ones

**How to add a new closure.**

The process of adding a new closure is quite laborious the first time because code has still not been optimized for it. It is necessary at least to

* Write the code of the new closure looking at the existing ones and adding an entry to the OntologyVocabulary class (the file java mandatory and possibly the OWL to remain synchronized)
* One modification in the code of functional chain to allow it to recognize the new closure name. The modification correspond to a copy paste of the code of an if with little changes in the strings called
* Two simple modifications in the Function Executor code

Probably points 2 and 3 can be removed in the final version and so the process of adding a new closure will correspond to writing its code and signaling at ontological level its existence. Also the file OntologyVocabulary.java can be obtained from the OWL ontology besides some detail that can be adjusted i.e. literal values section