

TDK-dolgozat

Németh András



EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS
PROGRAMMING LANGUAGES AND
COMPILERS DEPARTMENT

Data access optimization

Supervisor:

Tóth Melinda
assistant lecturer
ELTE-IK, PNYF

Author:

Németh András
computer science MSc
night school, II. class

Budapest, 2012.

Contents

1	Introduction	1
1.1	RefactorErl	3
2	Abstract Semantic Graph	4
2.1	Lexical Layer	5
2.2	Syntactic Layer	5
2.3	Semantic Layer	9
2.4	Mathematical Model	11
2.5	Specializing The Model	12
2.6	Graph Schema	13
2.7	Path Function	14
3	Graph Implementations	16
3.1	Graph Representations	16
3.2	Mnesia Based Graph	18
3.3	C++ Based Graph	20
3.4	Graph Abstraction	22
3.4.1	Requirements	22
3.4.2	The Startup Flag	24
3.4.3	Abstract Graph Layer	25
3.5	Kyoto Cabinet Based Graph	26
4	Improvement Areas	28
4.1	Performance Issues	28
4.2	Setting Goals	30

4.3	Performance Comparisons	31
4.4	Connecting Erlang and C++	32
4.4.1	Ports	33
4.4.2	Linked-in Driver	33
4.4.3	Erl Interface	34
4.4.4	C Node	34
4.4.5	Natively Implemented Function (NIF)	34
4.4.6	Chosen Setup	35
5	Abstract Semantic Graph Representation	36
5.1	Generic Graph Representation	36
5.1.1	Concepts	36
5.1.2	Graph Libraries	42
5.2	Abstraction From The Representation	42
5.2.1	Graph Schema	45
5.2.2	Path Function	46
5.3	Interpreting Path Expressions	48
5.4	Erlang Types	51
5.4.1	Type System	51
5.4.2	Erlang Term	52
5.4.3	Type Introspection	54
5.4.4	Recursive Data Structure	54
5.4.5	Type Precedence	55
5.4.6	Formal Definition of the Type System	56
5.4.7	Type System Object Model	57
5.4.8	Erlang Type Representation	62
5.5	Abstraction Layers	64
6	Storage Model	66
6.1	Persistence Models	66
6.2	mmap	66
6.3	Defragmentation	67
6.4	Kyoto Cabinet	68

6.5	Data serialization	69
6.6	Comparing The Persistence Models	69
6.7	In-memory Caching	70
6.8	Multithreading	71
6.8.1	Multithreading in Erlang	71
6.8.2	Readers-writer Lock	72
7	Conclusion	73
7.1	Future Improvements	73
A	Graph Measurement Commands	78
B	Source Code of Demonstration Applications	79
C	Commands Used for the Demonstration	89

Abstract

Efficient and fast applications require optimized and effective data representation. RefactorErl uses an abstract graph to represent Erlang source code. For processing large scale industrial applications the tool needs to perform operations faster than the current prototype implementations. Finding an efficient representation of the abstract graph to achieve faster data processing is the main purpose of this thesis. The secondary goal is to redesign the abstract graph implementation into a new structure.

Chapter 1

Introduction

A data structure is an actual implementation of an abstract data type, for instance graphs. These hide the actual data representation from the application and provide functions such as abstract graph operations. These ususally provide access to complex algorithms as well, for instance a shortest path search.

The abstract graph, which represents Erlang [11] source code in Refactor-Erl [19], is an out of the common model in the sense that it has application specific features and optimization for example a graph pattern matching algorithm for finding paths and subgraphs efficiently. Several graph implementations are created in the past few years, all of them had achievements and limitations. None of them proved to be effective enough to let all features of the tool to work on large project code bases ($> 1\text{M eLoc}^1$).

As the primary goal it is necessary to find an efficient graph representation to optimize the most expensive operations and to eliminate the limitations of current prototypes. The expectation is to gain performance. In the previous years new ideas arose to help overcoming performance issues and new data storages found performing more effectively than the ones currently used. Therefore it is desirable to define a new structure of loosely coupled components to provide a framework that can be easily adapted to the actual needs. This is the secondary goal of this thesis. The expectation is to gain flexibility:

¹eLoc = effective lines of code

in this structure if someone wants to change a component, he or she would be able to do it without harming the other components. Now it can be done only by writing a brand new graph implementation from scratch.

This thesis is divided into five main parts:

- Reviewing the formal model of the graph. It is necessary to know what we are going to re-implement (chapter 2).
- Review of the prototype implementations. These are called prototypes because each is special and different from the others (chapter 3).
- Study of opportunities. Review of general graph representations, and the possibilities of connecting Erlang and C++. (chapter 4).
- Finding improvement areas. Concrete implementation plans are written here describing the chosen representation as an abstract data type and the new graph database structure. Another topic is the modeling of Erlang types in C++ (chapter 5).
- Introducing storage models. Finally a brand new model is introduced along with storage related algorithms. The topic of multithreading takes place here as well. (chapter 6).

My contribution was started with the creation of the graph abstraction (section 3.4). Based on the concept I implemented a new graph database using Kyoto Cabinet [14] (section 3.5). As a prerequisite of the thesis I compared the graph implementations in performance point of view (sections 4.1 and 4.3). I searched for improvement areas and compared possible solutions (chapter 4). Finally as the main topic of the thesis I defined data structures and algorithms that are meant to optimize performance and data access in the system, namely: generic graph representation (section 5.1), abstraction from the representation (section 5.2) - to allow the implementation of operations and algorithms such as the path function. Introduced an efficient way to interpret graph schema (subsection 5.2.1) and path expressions (section 5.3) and created a C++ model of the Erlang type system (section 5.4). I defined persistence models (chapters 6.1, 6.2, 6.4), a defragmentation (section 6.3)

and a serialization method (section 6.5), in-memory caching (section 6.7) and a multithreading model (section 6.8).

1.1 RefactorErl

Today code refactoring and comprehension became very important in software engineering and maintenance. Static source code analyzer tools enable developers to understand big parts of a program code in less time. By providing static semantic information about source pieces, code understanding is a lot easier and straightforward today, compared to the common techniques², as well as doing code restructuring became fast and safe. RefactorErl is a good example of code comprehension tools, this is an open-source static code analyzer and restructuring framework developed at ELTE University [21] [30]: it helps programmers to do code comprehension and legacy code refactorings in an easy way. This tool operates on Erlang source files: first the user loads these into the tool, which then constructs the abstract syntax tree from program text. Afterwards it searches semantic information with the help of built-in semantic analyzer processes. In the past years many developers chose to use this tool because it provides a feature-rich environment that makes the development of Erlang programs easier.

²e.g. by using the 'grep' Unix command to find function call references. The results are always ambiguous and contain only a subset of the full possibilities.

Chapter 2

Abstract Semantic Graph

In this section the mathematical model of a generic semantic graph is defined as an abstract data type. Reviewing and analyzing the different layers of Erlang source code will point us to the definition of the semantic graph that is capable of representing the syntactic and semantic structure of program text.

Source code processing Erlang source files are processed in three main steps. As the first, the text is tokenized by the lexical analyzer (scanner). As an intermediate step the token representation is further processed by the pre-processor to let the syntax analyzer (parser) work on the compilation-ready version of the program text. The output of these processes is the abstract syntax tree (AST); this is the basic representation. In the third step, semantic analyses go through the AST, each processing a subtree, and inserting semantic information into that AST. One can visualize this representation as a graph where labeled nodes are pieces of the Erlang program (tokens, syntactic and semantic nodes) and the labeled links are the connections between them. The edges are defined by the syntactic and semantic rules of the programming language. In the following the mathematical model is introduced by describing the three sub-models used by the three-step code processing. In the last subsection these are put all together to represent source code by the generic semantic graph.

2.1 Lexical Layer

The role of the model is to represent the exact contents of Erlang source files character by character (including whitespace). It is necessary to be able to restore the source file into its genuine format because of the transformations (refactorings). To let these work properly the source code needs to be modified and saved. The lexical information is stored in nodes having the class *token*. Every *token* node contains a single Erlang token, together with surrounding whitespace and comments. After the first step of code processing, the source file is represented with a list of token nodes chained together with links tagged as *next*.

Preprocessor support In Erlang the programmer may use macros. Macro definitions can contain a valid Erlang expression and also can have parameters. Usually these are stored in header files which can be included in source files. The lexical part of code processing cannot be made complete without the following actions: inserting contents of the header files into the source file and substituting macro definitions. The Erlang compiler processes source files in this way, thus the resulting bytecode cannot be restored into the original source code form. On the other hand executing these actions may result in data loss, for RefactorErl both the genuine and the transformed text is needed, so it needs a different approach: a transformation layer is inserted between the lexical and syntax analysis. It passes on the appropriate tokens (inserts text of included files and substitutes macros where required) to the parser, and at the same time it builds graph parts that facilitate recovering these later. At syntax analysis, a parse-tree is constructed with the transformed tokens.

2.2 Syntactic Layer

The lexical graph representation is embedded in the syntactic structure of the graph. The syntactic elements of Erlang are represented by four node classes, which symbolize the element blocks of Erlang language:

- *file*: logical entity, represents an Erlang source file module or header
- *form*: building blocks inside an Erlang module. There are two types: module attributes and function definitions.
- *clause*: expression sequences, such as function arguments, clause patterns and guard expressions. Clauses always introduce visibility areas.
- *expr*: expressions. There are three types, such as patterns, guards and expressions.

To understand the basics of Erlang syntax, an example is written to help visualize the full Erlang source code structure with the possible nodes and links. The example module which is analyzed follows:

```
-module(test).  
-compile(export_all).  
  
double(A) ->  
    A + A.
```

The full graph representation of the example Erlang module is depicted on figure 2.1.

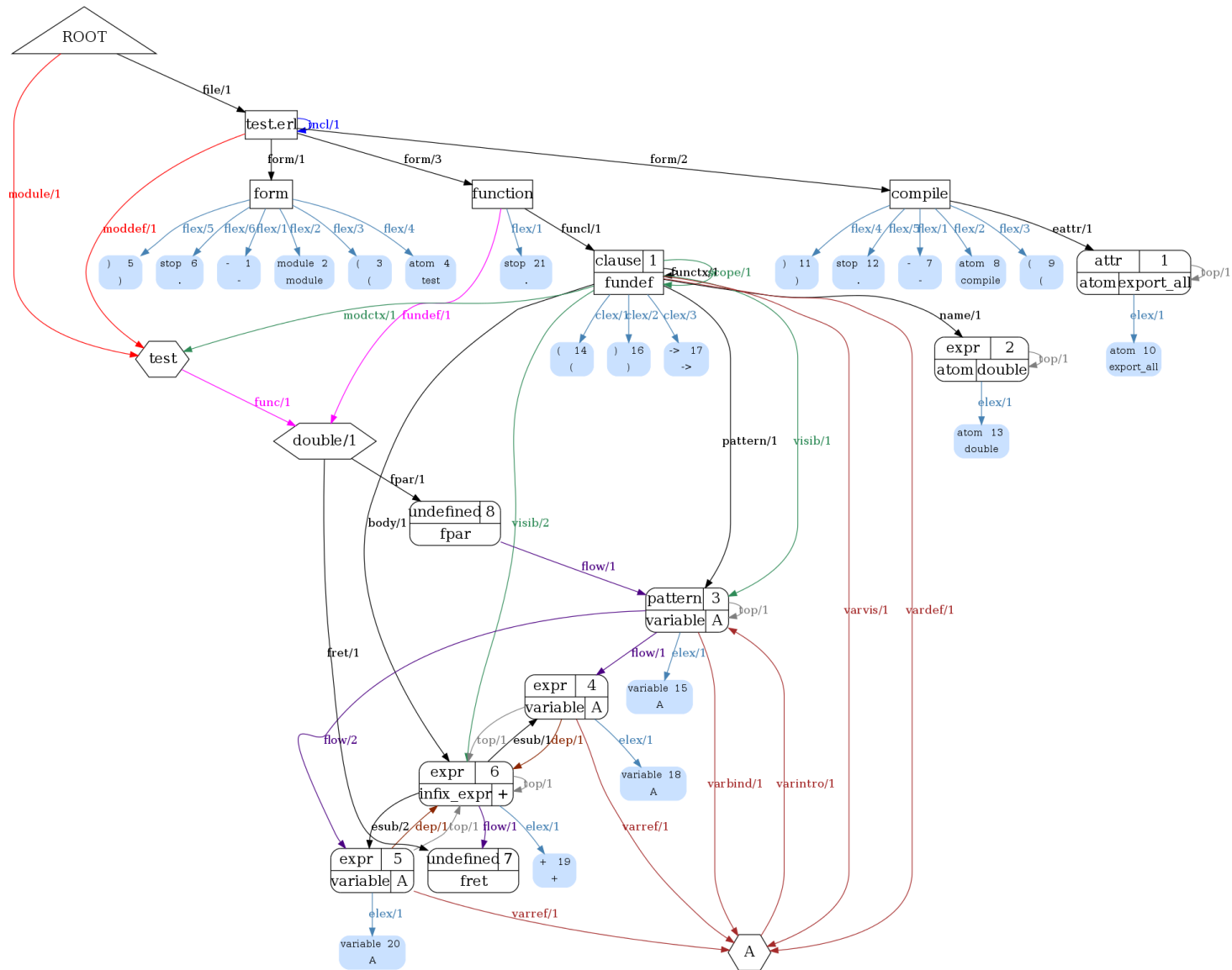


Figure 2.1: Example graph

The structure of the AST is described as follows.

- The top node of the tree is called the “root node”.
- Every loaded source file has a *file* tag link with the root.
- File nodes are linked to their top level form node with a *form* tag.
- The attribute nodes are linked to their contents with the *attr* tag.
- Function nodes representing each function in a module are linked to the clauses with the *funcl* tag.
- Since Erlang function heads can contain patterns, the clause nodes may have attached *pattern* expression nodes.
- Clauses also may have *guard* expression nodes.
- Function clauses have a *name* expression node (it is always has the kind atom).
- Clauses are linked to their expressions with a *body* tag.
- The **after** clause of the **receive** expression has a timeout expression, the clause is linked to it with a *tmout* tag.
- Expressions are linked to their direct subexpressions with a tag *sub*.
- Compound expressions (**if**, **case**, **receive**, **fun**, and **try**) are linked to their main clauses with an *exprcl* tag.
- The contents of **catch** and **begin ... end** expressions are also modeled with a single *exprcl* clause.
- The **after** clause of the **receive** and **try** expressions is linked with an *aftercl* tag.
- The handler clauses of the **try** expression is linked with a *handlercl* tag.

As written earlier, the syntactic model with the embedded lexical graph are representing the abstract syntax tree of the source code.

2.3 Semantic Layer

The semantic model is built on top of the AST. It consists of node and link types. The semantic layer includes information about defined modules, functions, variables and indirect structural links between syntactic units.

Structural links. Besides the direct syntactic links that represent the abstract syntax tree of the source code, the following structural links are created:

- Every expression is linked to its topmost super-expression with a *sup* tag.
- Every scope clause is linked to its every directly or indirectly contained clause with a *scope* clause.
- Every compound expression that contains clauses is linked to its every direct clause (in proper order) with a *clause* tag.
- Every clause is linked to its every direct subexpression with a *visib* tag. Note that these are the expressions that are accessible from subexpressions with a *sup* tag.

The last two links may seem redundant, as they duplicate direct syntactic links. Their primary purpose is to make syntax tree traversal easier by defining a simple expression structure. *Visib* links also make handling variable visibility areas easier.

Module information. Every module that is known to exist in the program (that is, the definition of the module or a construct that refers to the module is found) gets a semantic node. The class of this node is *module*, and its only attribute (*name*) is the name of the module.

The following links are created for a module:

- The root node is linked to the module node with a *module* tag.

- When the module definition is found, the module is linked to the defining file node with a *moddef* tag.
- The module node is linked to every scope clause node that is defined in the module with a *modctx* tag. This is used to track the module context of expressions in the module.
- Every expression that explicitly refers to the module is linked to the module node with a *modref* tag.

Function information. Every function that is known to exist in the program (that is, defined in a module, or a reference to the function is found) gets a function node. The class of the node is *function*, and it has two attributes: *name* and *arity*, the name and the arity of the function. The attributes are stored in the node record.

The following links are created for function nodes:

- The module node that contains the function definition is linked to the function node with a *func* tag.
- When the definition of the function is found, the function node is linked to the defining top level form with a *fundef* tag.
- When a module exports a function, the module node is linked to the corresponding function node with a *funexp* tag.
- When a module imports a function, the module node is linked to the corresponding function node with a *funimp* tag.
- Every expression that explicitly refers to the function is linked to the function object with a *funref* tag.

Variable information. Every variable in the program gets a semantic variable node. The class of this node is *variable*, and its only attribute (*name*) is the variable name.

The scope and visibility rules for variables are much more complex than for functions, thus the semantic links are more complex. The following structure is described by semantic links:

- Every variable has a *scope*, a program part that is affected by the presence of the variable. The scope of a variable is defined by a function clause or a list comprehension. These are modeled with a clause node that has its scope attribute set to true.
- The subtree of a variable scope contains bodies (sequentially executed expression lists, modeled with clause nodes). The variable is visible in such a clause when there is an expression in the list that introduces the variable (binds it to a value) or when the variable visibility is inherited from an outer clause. In the former case, the variable is visible from the expression after the introduction until the last expression, and in the latter case, it is visible in every expression.

This structure is described by the following links:

- The scope clause node is linked to every variable node that is defined in it with a *variable* tag.
- Every clause node is linked to every variable node that is visible in with a *varvis* tag.
- The variable node is linked to every top level expression that introduces the variable to a clause with a *varintro* tag.
- Every variable expression that can bind the variable to a value is linked to the variable node with a *varbind* tag.
- Every variable expression that reads the value of a variable is linked to the variable node with a *varref* tag.

2.4 Mathematical Model

This section describes the mathematical model that can represent the previously defined layers (sub-models). It is basically a directed, labeled graph.

The raw mathematical model is defined here:

$$\mathcal{SG} = (N, A_N, A_V, A, T, E),$$

where

- N is the set of graph nodes, these will represent the nodes of the syntax tree and additional semantic nodes,
- A_N is a set of attribute names,
- A_V is a set of possible attribute values,
- $A : N \times A_N \rightarrow A_V$ is the node labeling partial function,
- T is a set of edge tags, and
- $E : N \times T \times \mathbb{N}_0 \rightarrow N$ is a partial function that describes labeled, ordered edges between the nodes.

2.5 Specializing The Model

In the previous sections three sub-models were introduced which are capable of representing Erlang source code structure (syntax) and semantics. First of all each type of node is classified in the graph since these hold different number and kind of data. Therefore these can be modeled with different data descriptors. For example a token node stores the string representation of the token along with the surrounding whitespace, while a file node stores the path and last modification date. We need to define which class of nodes have what type of link tags too. Building these relations into our graph as restrictions enable us to create an efficient implementation and increases the accuracy of the representation (we forbid certain kind of links and forbid invalid node classes). On the other hand it also enables us to design an effective, yet specific graph traversal algorithm since we know in-place how to search for specific language elements. If these constraints are built into the graph, the result would be an abstract semantic data type. To preserve the genericity of

the formal model, we may define a *schema* structure which can describe such relations. Based on the schema, we can define an optimized graph traversal algorithm. These are described in the following.

2.6 Graph Schema

Definition:

$$\mathcal{SCH} = (C, N_c, A_c, E_c),$$

where

- C is the set of permitted node class names.
- $N_c : N \rightarrow C$ is a total function that classifies nodes.
- $A_c \subseteq C \times A_n$ is a relation that contains the attributes that are used for a given class of nodes.
- $E_c : C \times T \rightarrow C$ is a partial function that describes valid edge tags between node classes.

A given graph $\mathcal{SG}$ is valid if the following applies with the schema $\mathcal{SCH}$:

- $\mathcal{D}_A = \{(n, a) \mid (N_c(n), a) \in A_c\}$ the attributes of a given node is the same defined for its class in the schema
- $\forall n, t, i : N_c(E(n, t, i)) = E_c(N_c(n), t)$ all links created are the ones permitted by the schema.

The restrictions introduced here can be applied during graph creation and transformation. By injecting the schema into the graph and using it while creating nodes and links, data will remain consistent and reliable. The schema makes the graph to be semantic, since it knows the meaning of its elements. The definition of the *path* function is also possible, that can traverse the graph using the information defined in schema.

2.7 Path Function

Since the structure of the graph has a fixed depth, information can be efficiently extracted by a breadth-first search algorithm. The path function is one of this kind. In addition, it takes a *path expression* parameter, which describes the steps of graph traversal. These expressions define a query language that makes it straightforward to traverse graph structures with fixed depth. *Path expression* definition:

$$\begin{aligned}\mathcal{P} &= [PE_1, PE_2, \dots, PE_k] \\ PE_i &= (t_i, d_i, f_i)\end{aligned}$$

where

- $t_i \in T$ is a link tag.
- $d_i \in \{F, B\}$ is a direction specifier (F stands for forward and B stands for backward).
- $f_i : \mathbb{N}_0 \times N \rightarrow \mathbb{L}$ is a filtering function

Input of the path function needs to be a set of nodes, but at least one. The function may be called recursively, evaluating one expression in one iteration. The output of the function is also a set of nodes that are the resulting node set to where the links with that label from the input nodes are pointed. The function terminates when either there are no more nodes in the set or if there are no more *path expressions* to evaluate. Path function definition follows:

$$\begin{aligned}\mathcal{E}(n, []) &= [n] \\ \mathcal{E}(n, [PE_1, PE_2, \dots, PE_k]) &= \mathcal{E}(\mathcal{E}(n, PE_1), [PE_2, \dots, PE_k]) \\ \mathcal{E}([n_1, \dots, n_k], P) &= \mathcal{E}(n_1, P) + \dots + \mathcal{E}(n_k, P) \\ \mathcal{E}(n, (t, F, f)) &= \{n' \mid n' = E(n, t, i) \wedge f(i, n') = \text{true}\} \\ \mathcal{E}(n, (t, B, f)) &= \{n' \mid E(n', t, i) = n \wedge f(0, n') = \text{true}\}\end{aligned}$$

The path function can be implemented as a higher-order function which allows us to define any type of filtering functions that iterate over the set of nodes. The current implementation has the following filtering functions:

- TrueFilter: $true(i, n) = true$,
- IndexFilter: $index(j)(i, n) = (i = j)$,
- RangeFilter: $index(b, e)(i, n) = (b \leq i \wedge i < e)$,
- IntersectionFilter: $intersect(n', t')(i, n) = (n \in \mathcal{E}(n', [(t', F, true)]))$,
and
- AttributeFilter: $attribute(a, \odot, v)(i, n) = (A(n, a) \odot v)$, where $\odot$ can be $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$.

Additional helper functions (which are not filters in themselves):

- $last(n, t) = \max\{i \mid (n, t, i) \in \mathcal{D}_E\}$
- *and*, *or*, and *not* logical operations for the *attribute* function

Chapter 3

Graph Implementations

This chapter describes the different graph representation techniques in general, additionally presents the current implementations and the different graph databases already developed.

3.1 Graph Representations

The graph is an abstract data type: in one structure it contains edges and vertices, plus the algorithms and operations defined on them. Per se, neither edges nor vertices complex, but the concrete structure built by these elements can be very difficult to understand and maintain consistent. The graph is basically a representation of objects and binary associations between them. In general graphs are suggested as popular modeling tools because of its simplicity, manageability and expressiveness [29]. There are a few techniques describing how to represent and store graph data in a database [13].

1. Adjacency list

- Vertices are represented by objects or records
- Each vertex has a list of its edges to adjacent vertices
- This representation allows the storage of additional data on the vertex
- The graph traversal is relatively slow

- Deleting a vertex or edge we need to find all adjacent edges. Generally the graph transformation operations are relatively cheap.

2. Incidence list

- Both vertices and edges are represented as individual objects or records
- Each vertex stores its incident edges
- Each edge stores its incident vertices
- Provides more flexibility
- This representation allows the storage of additional data on both vertices and edges
- Graph traversal is relatively slow
- When deleting vertex or an edge it is necessary to find all adjacent vertices, edges, generally the graph transformation operations are relatively cheap

3. Adjacency matrix

- It is a two-dimensional matrix
- Rows represent source vertices
- Columns represent destination vertices
- The cost of one edge is stored in the matrix
- Other data need to be stored externally
- Graph traversal is relatively fast
- Deleting a vertex or edge the matrix needs to be resized which is relatively slow

4. Incidence matrix

- It is a two-dimensional matrix of `boolean` values
- The rows represent vertices

	Adjacency list	Incidence list	Adjacency matrix	Incidence matrix
Storage	$O(V + E)$	$O(V + E)$	$O(V ^2)$	$O(V \cdot E)$
Add vertex	$O(1)$	$O(1)$	$O(V ^2)$	$O(V \cdot E)$
Add edge	$O(1)$	$O(1)$	$O(1)$	$O(V \cdot E)$
Remove vertex	$O(E)$	$O(E)$	$O(V ^2)$	$O(V \cdot E)$
Remove edge	$O(E)$	$O(E)$	$O(1)$	$O(V \cdot E)$
Adjacency query	$O(V)$	$O(E)$	$O(1)$	$O(E)$

Table 3.1: Time complexity comparison of different representations

- The columns represent the edges
- Data of the vertices and edges needs to be stored externally
- Graph traversal is relatively fast
- Deleting a vertex or edge is relatively slow (matrix needs to be resized)

The required graph representation from the above list depends on the nature of algorithms to implement. Fortunately the graph is an abstract data type, its internal data structures are accessed only by its self-defined algorithms. The time complexity cost of basic operations are summarized in table 3.1.

When removing edges or vertices on an adjacency list, finding all adjacent vertices or edges is needed, which can take time. The adjacency matrix is slow on adding or removing vertices (need to resize matrix), while the incidence matrix is slow on adding and removing both vertices and edges (need to resize matrix).

3.2 Mnesia Based Graph

This graph representation is an incidence graph. Mnesia [18] is a distributed relational DBMS written fully in Erlang. The vertices and edges are stored as records in individual tables. This implementation assumes that there is a

special node class called *root* which has no attributes. There is only a single node of this type, called the *root node*. The goal of the implementation is to evaluate path expressions efficiently: as the records themselves are Erlang tuples, it allows searching records not only by id, but with wildcarded patterns. Mnesia is persistent, data is stored on disk but is fully cached in memory at the same time. It also allows parallel queries, because it locks the record that is being accessed, so by this way it eliminates clashes. The first graph implementation is based on Mnesia, hence it's name.

The implementation has a few disadvantages. It is developed to be a distributed database for soft real-time applications, so it has an extended locking mechanism for consistency safety which is unnecessary here, because the tool stores data only on the local Erlang node. In summary, Mnesia was not proved to be an efficient implementation. It's structure is visualized on figure 3.1.

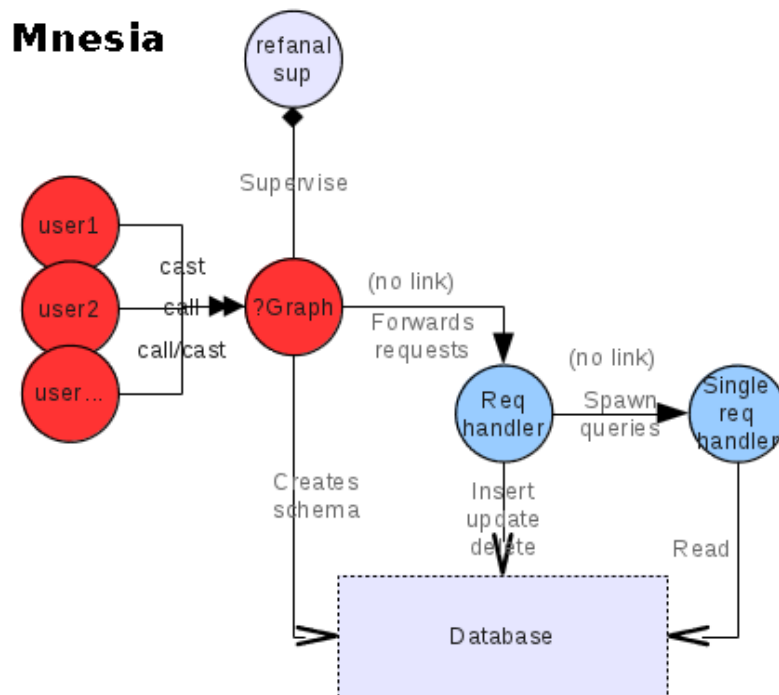


Figure 3.1: Mnesia implementation

The processes accessing the database are connected directly to the Refac-

torErl graph implementation, the Mnesia database. This allowed no flexibility, it was not possible to try different approaches. The `graph_server` - denoted by ?Graph on the picture - forwards the requests to `req_handler` that spawns a parallel process for non-transforming operations, else it waits for all ongoing transactions to complete before starts processing the transforming query individually. Until the transforming query is not finished, all others are queued. This way a readers-writer lock is implemented here on the graph level.

Limitations There is no other database written in Erlang/OTP which is suitable for using as a persistent storage. But its distributed nature comes with the price of the relatively slow operations. The cost of graph operations proved to be very expensive. Other problem is the size of database. Mnesia stores persistently everything on hard disk, but for faster access everything is loaded into memory also. Building a graph from industrial-size code resulted in memory shortage. And at least, the greatest disadvantage was that it was strongly connected to other components of the tool, that made hard to change the implementation.

3.3 C++ Based Graph

The implemented representation is the adjacency list. Considering the limitation of Mnesia graph, the C++ based graph was built to achieve a more efficient, faster and more optimal graph database. Here, the graph data structure is implemented in C++, with the persistence layer also, but a portion of the graph operations remained in Erlang, such as path and schema handling. The loading source code means instantiating classes in memory, so those can be accessed faster and the result is a more efficient graph traversal. Data stored mainly in memory but it is also written to a file after user operations to assure no data loss. Because the data is stored mainly in the memory - despite its relatively lower memory consumption compared to Mnesia - memory shortage is a problem here too.

There are a few techniques to connect C/C++ and Erlang. For this task,

Natively Implemented Function (NIF) [10] was selected. As its name pretends, it acts like an arbitrary Erlang function, can be called from Erlang code as an ordinary function, but in the background the virtual machine calls the actual function in a Unix shared library or dll. This is available from R14B release only. This way there is no possibility for parallel queries because NIF calls are not prepared for multithreading, these are strictly sequential. The storage is visualized on figure 3.2.

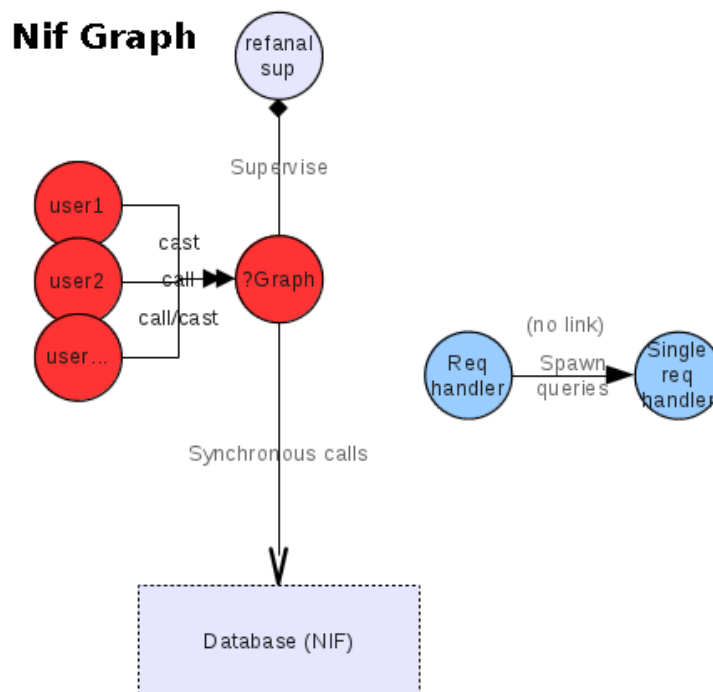


Figure 3.2: NIF implementation

The `graph_server` directly uses the database. Big advantage is that it is notably faster than its predecessor, however the database is strictly single threader, without parallel query execution. Although the request handler process is not removed but not used at all here.

Limitations The graph is implemented in C++, but the path function in Erlang, except that the forward or backward step in the graph is made in C++. The path expression evaluation and the filters are executed in Erlang.

Additionally the schema handling is still on the Erlang side. Another limitation is that all of the database operations are executed sequentially, because only the graph server process (?Graph) can serve them. The order of operations are determined by the order those are inserted into the graph server process's mailbox.

3.4 Graph Abstraction

The first prototype of Nif graph interchanged its predecessor as a whole new graph database layer, bound in implementation to other parts of the tool. RefactorErl was still not prepared to be used on top of multiple database types. The development of the tool became hard because two distinct branches had to be maintained, mapping all changes in one branch to the other, which holds the possibility of additional bugs. This is inefficient. Separating the abstraction from the implementation was needed hence to create a database abstraction. Additional improvement of the abstraction is that it makes possible to develop more implementations without changing the abstraction.

3.4.1 Requirements

Important requirement is that the upper level layers need to use the same abstract graph and not the actual graph implementation. Therefore the first step is the definition of the interface to allow the clean separation of layers. Important to mention that here we deflect from the OTP principle ¹ because the interface module will not have a dedicated server process. The usage of this principle results in strongly coupled components, modules tied strictly to each other with unnecessary dependencies. By creating an interface module, which exposes the abstract interface only, the two layers can be separated by hiding the database implementation from the upper layers. The most important is to define the required input and output parameters of the interface functions.

¹The OTP principle recommends organizing the code executed by the same process into one Erlang module.

The second requirement is to extract the common operations of the different database implementations. It allows the programmer to deal with the specific details of the implementation only. Separating the caller and the serving processes is also important to ensure that the layers are separated in space. On the upper layers there is no exception handling, so at the database level those are handled. This way the responsibilities are also separated. The inter-process communication are the most common task to be generalized. Secondly the schema handling can be exalted, all operations shall be independent of the schema. Thirdly the graph traversal algorithm, the path function is to be generalized: only one step on forward or on backward edge is needed to implement without the higher level logic. But on the other hand it is important to allow the programmer to implement an own search algorithm if it is needed.

As the third requirement it is necessary to allow the user to choose the database implementation to be used. This allows the user to run the tool on top of multiple databases. To achieve this, the entry point of the databases shall be the same. In Erlang we can use dynamic name look up in the form of `module:function(Par1, ..., ParN)` calls, where both the module and function can be looked up dynamically, so it can be set as an initial parameter by the user. The easiest is to define an initialize flag for the Erlang virtual machine in the command invoked to start it. This flag shall indicate the name of database module to be used.

The reason of the above described interface separation and common operation generalization is to allow the database developer to deal with the implementation part only. With coding the primary graph operations we will get a working database layer and do not have to care about the method how to connect it to the application layers. In this way the tool is able to run on top of different database implementations without recompiling the tool and also it is possible to choose how and where to store the graph.

3.4.2 The Startup Flag

The two graph implementations - Mnesia and C++ graph - are made independent from application layers by defining the abstract graph interface. The interface module is the entry point for all upper layer applications and they do not need to know what is the underlying implementation, it is well hidden now. The common operations in a generalized form were moved into a `behaviour` module. This can be called as a general - or generic - graph storage module. This implements the operation handling logic, but not the concrete execution. The name of this behaviour is `refcore_gendb`. It just calls one callback function on the concrete storage module passing on the name of the concrete operation to be executed. At startup the supervisor calls `refcore_gendb` (on the figure it is called *refdb*) and it starts up itself with calling the `init` function in the concrete storage module *refdb_impl*. So it was needed to separate the two concrete storages into two separate Erlang modules; the `refdb_mnesia` and `refdb_nif`. The `refcore_gendb` module can use both of the concrete storages. The graph implementations are using their own storage directly. The new setup of the graph subsystem is showed on figure 3.3.

The startup scripts of the tool are modified too to pass a new flag to the Erlang virtual machine when invoking the startup command. This flag is accessible from anywhere in the code. There is a need for two flags: `-dbmod` which is the name of the concrete storage module and `-dbargs` which sets the additional startup parameters passed to the concrete module if needed. For example to use Mnesia these are the following: `-dbmod refdb_mnesia -dbargs []`. Database selection is simplified in the startup script. Only the `-db` has to be set with the name of the database to use. It can be `-db mnesia` or `-db nif`. This parameter is recognized and substituted by the startup script to the appropriate `-dbmod` and `-dbargs`. In this way it became possible to use the tool on top of different databases.

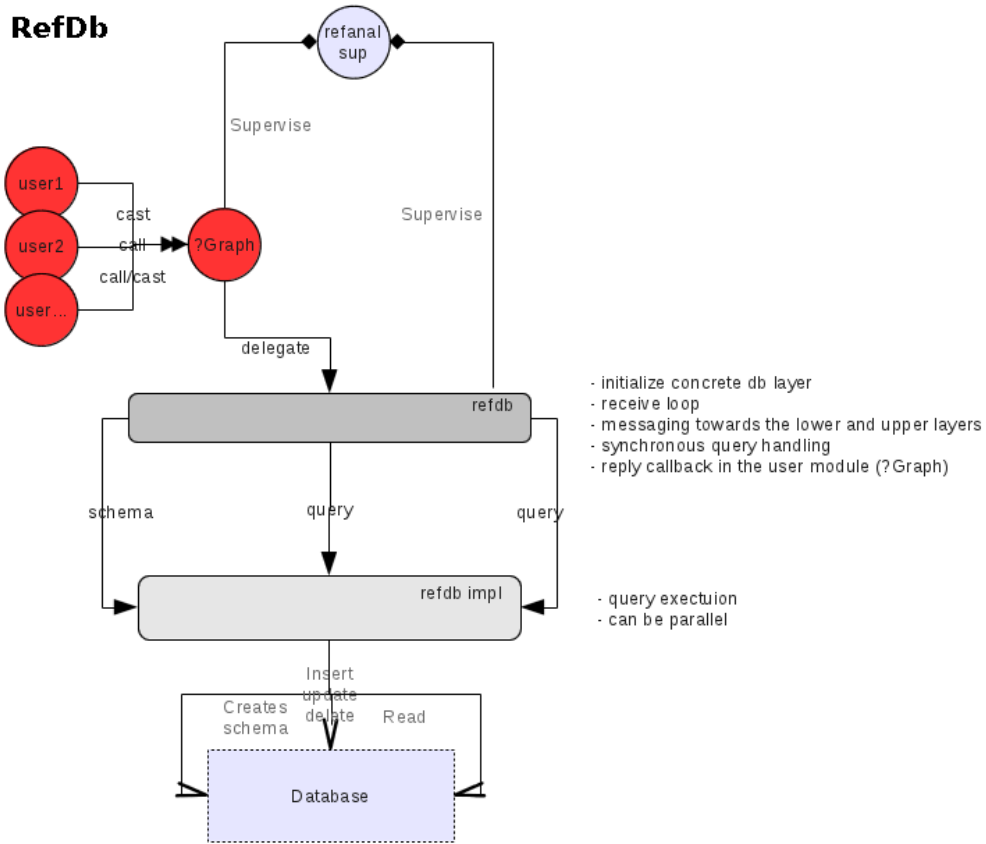


Figure 3.3: Graph subsystem supporting the startup flag

3.4.3 Abstract Graph Layer

At this point, the two graph databases are moved under the same interface but the abstraction is not yet completed. The next step is to call the concrete operation as a callback instead of calling one general function on the concrete storage module. Therefore if a function exists in the interface, it must exist in the implementation. This way the interface directly defines the implementation. Another task is to raise the schema handling to the general graph module. This way the database developer does not need to implement other tasks than the primary graph operations. The second general part is the path function which is also moved here with but the possibility is retained to implement a custom path function if needed. As a minimum for the path function, the one-step moving has to be implemented by the developer. This

kind of setup is shown on figure 3.4.

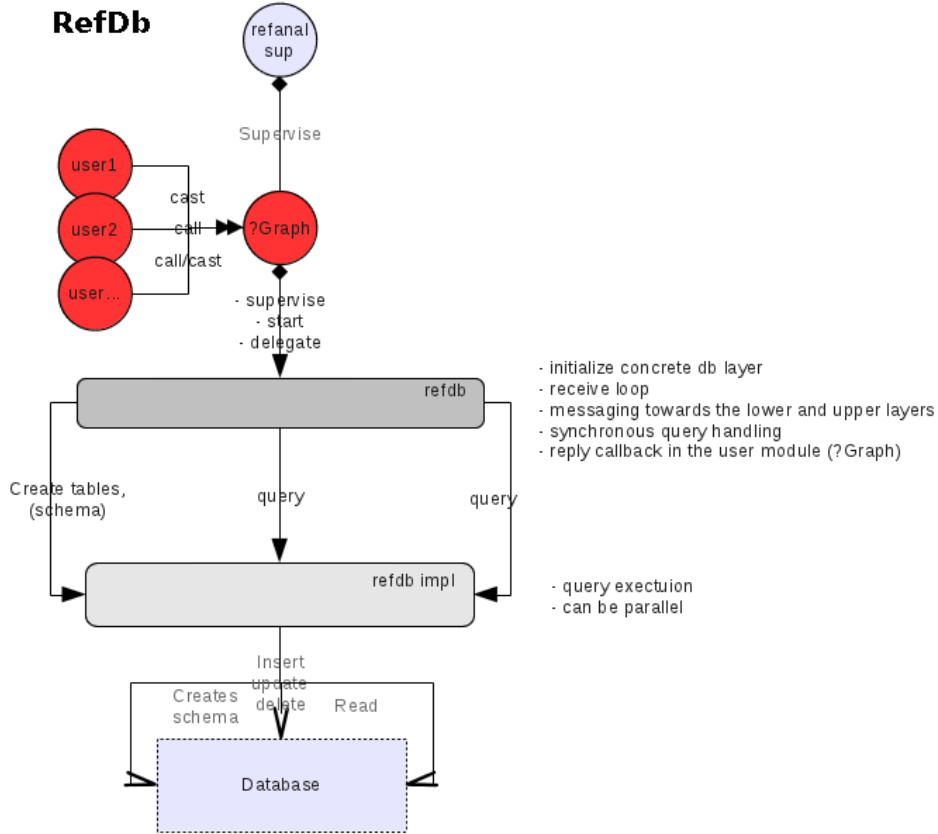


Figure 3.4: Graph subsystem separated by the abstract layer

Only one more thing is changed: The supervisor does not start two server processes, instead it calls to the original *?Graph* module which will start the initialization of the graph server process.

3.5 Kyoto Cabinet Based Graph

The two databases described above could not deal with the problem of memory shortage. There is a need of a storage model that stores the graph primarily on disk and caches the most commonly used data in memory. Kyoto Cabinet is a C++ based DBMS which is created with efficiency in mind. With this we can use a hash table or a B+ tree with MurMurHash 2.0 [27] as

the hash function for fast look up times. Also data is stored primarily in a file and the file is mapped into memory with the `mmap` [17] call. Because of the hashing the look up time is constant, and if a B+ tree is used it is not worse than $O(\log n)$ [14]. There are hash table implementations in C++ but Kyoto Cabinet is special because it is designed to handle enormously large amount of data if needed (max. 8 exabyte) without performance degradation. The constant look up time is important because the most of the graph operations are random data retrievals and the graph transformation are less frequent. Another fancy feature is that it is fully customizable: size of memory cache, record alignment, bucket size, etc. can be set when the database is opened. The memory cache size can be customized as well. The main part of the graph implementation remained in Erlang and Kyoto Cabinet is used as a key-value store for persistence. A minimal C code part is written to be able to interact with Kyoto Cabinet. First time the data is passed to the C functions as strings, but it proved to be inefficient. The conversion from Erlang expressions to strings and vice versa is very slow. It is changed to use the Erlang built-in serialization using the `binary_to_term` and `term_to_binary` functions. The memory footprint of the tool using the default database parameters is around 400MB so memory shortage is not a problem anymore. It is also possible to fine tune Kyoto Cabinet by setting the `-dbargs` parameter manually, in the form of a proplist: `[dbtype, '+', apow, 2, ...]`. The elements of the proplist must be the names defined by the Kyoto Cabinet `kcdbopen()` [15] function's parameters. The explicit setting of the `-dbtype` parameter is also required:

```
$: bin/referl -dbmod refdb_kyotomini -dbargs "[{dbtype, '+'}]"
```

Limitations Still there is no parallel query execution, the operations are handled sequentially. Moreover, since only the most basic operations are moved to the C side such as `create`, `update`, `data`, `delete`, `exists` and `delete_all`, managing graph links and paths will do more than one call into the C/C++ driver which is very inefficient knowing that every driver call has a processing overhead.

Chapter 4

Improvement Areas

4.1 Performance Issues

This year the tool was introduced in large industrial projects. We have loaded more than 4 million eLoc code into the tool. The most general features work relatively fast as expected, but we have seen a significant performance loss in certain operations; complex actions, for instance finding call chain graphs, dynamic function flow and data-flow analysis, these performed very slow. In general, it can be said that the tool is not fast enough for industrial use.

These charts 4.1 and 4.2 show a comparison of prototype implementations. The commands used for the measurement are enclosed in appendix A. Based on the values of the table it can be assumed that implementations in C++ perform better: while Mnesia is fully in Erlang, Kyoto graph has the storage in C++, it performs better, but Nif graph seems to be the most efficient from these three, where the main parts of the graph is implemented in C++. We can assume that the more of the graph database is made in C++ the more is the performance gain. Additionally, it can be seen from the CPU usage chart 4.2 that the implementations which do a sequentialization of graph operations create a bottleneck (Nif and Kyoto). Mnesia has a readers-writer lock implemented on the graph layer allowing the graph transformation operations to run sequentially. This turns out that by allowing fully parallel execution the performance can be improved. Also the memory con-

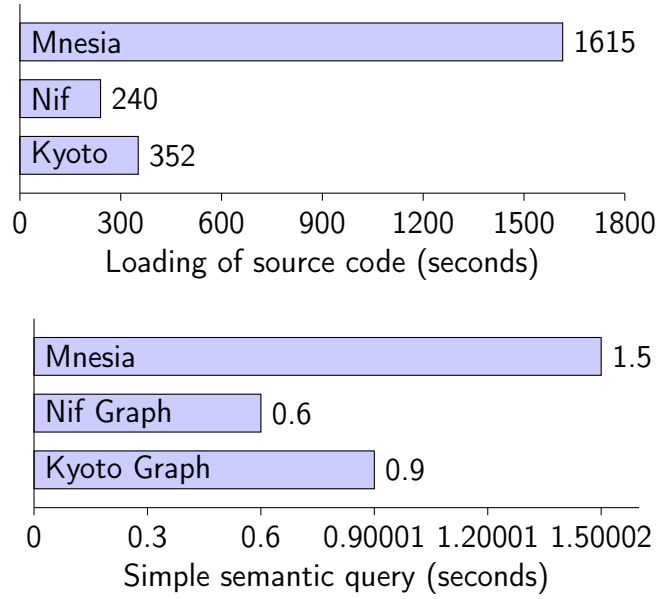


Figure 4.1: Time consumption of high-level operations

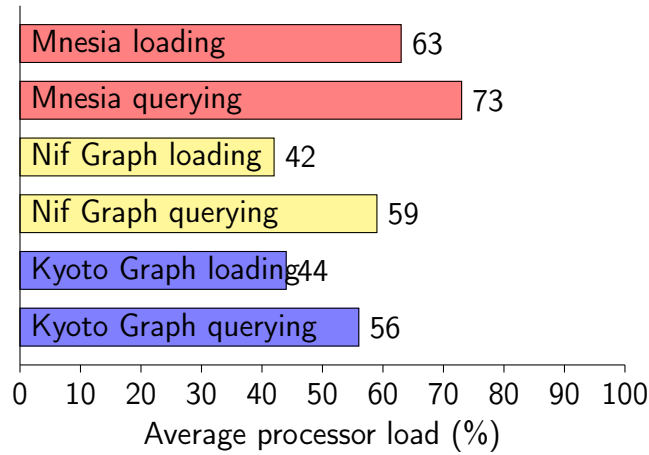


Figure 4.2: Processor utilization

sumption can be lowered if we use Kyoto Cabinet’s built-in caching with a relatively small performance loss. We believe that a well designed data model and its efficient implementation will help us to overcome performance issues, and enable us making an overall improvement in efficiency.

4.2 Setting Goals

Based on the experience gained from the graph implementations described in the previous chapter some conclusions can be made. First of all, we can say that by changing the graph layer of the graph database from Erlang to C++ we gain performance. If we analyze the C/C++ implementations deeper, we can see that some of the graph operations require more than one C/C++ function calls. If we could make the C++ function calls at the entry point of the graph operations we can eliminate some of the Erlang C++ interface overhead. This means that all graph operation need to be fully implemented in C/C++. This concept is shown on figure 4.3.

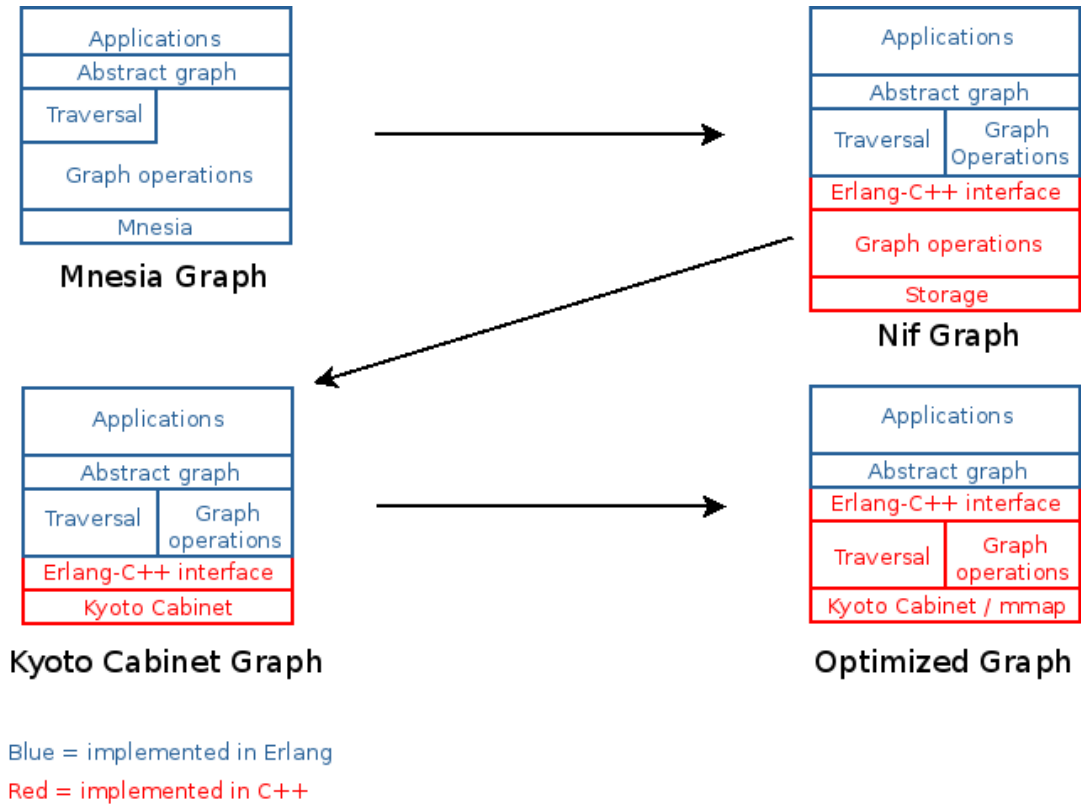


Figure 4.3: Parts of the graph in Erlang and C++

This shows the change of the Erlang/C++ borders. With red the C++ parts are shown and with blue, the Erlang. The last graph is the concept of the new implementation. This requires storing and interpreting dynamic

data coming from the Erlang VM. This problem domain is described in 5.4. Moreover a new implementation can be created, based on a high performance storage model. The new representation 5 may be made generic to allow to change to a new storage model anytime without writing a completely new graph database. Another important topic where optimization can be made is the full support of multithreading. As mentioned earlier, the graph operations are aligned by the time they arrive to the graph handler process's mailbox. The multithreading support is described in details in 6.8.

Compile time optimization Another aspect needs to be considered. A recent benchmark showed that llvm can do compile time optimization better than g++ and sometimes cannot [1]. But clang claims that their compiler is far better [4]. After looking into different benchmarks, it can be seen that sometimes the gcc produced binaries perform better and sometimes clang's. It can be assumed that the performance gain is application-specific. For a reference I would like to measure if it performs better in graph traversals in the next section.

4.3 Performance Comparisons

Erlang or C++ I would like to point back to the end of the previous chapter where the prototype implementations are compared. There, it can be seen, that C++ implementations do much better in performance. Also C++ provides greater flexibility while possibilities in Erlang are very limited, especially when it comes to memory management. I would assume that creating an application specific graph implementation in C++ would result in much better performance than using the Erlang built-in possibilities.

Benchmark Two simple graph traversal applications are introduced to confirm the assumption of creating more efficient C++ graphs compared to graphs in Erlang. The first one is created fully in Erlang using the *digraph* [8] library. A portion of the second one is also in Erlang, but the graph and the path search is in C++ using LEMON [16] graph library. Both of

the graphs are built up using the same source file: *graph.txt* containing the vertices and edges. The source code of both implementations can be found in appendix B and the commands used for the comparison are at appendix C. For the C++ graph, the NIF Erlang C++ interface (subsection 4.4.5) and also the Erlang type representation (subsection 5.4.8) was used (described in details in section 5.4. Both the interface change and the data conversion to represent arbitrary Erlang data generate a necessary computation overhead. The C++ graph is compiled both with g++ and clang++ to measure the difference. In Erlang the fastest graph implementation is *digraph*, that is why it was chosen. Both demonstration applications create the graph in memory and starts path searches between the root and the other nodes multiple times parallelly. The result of the benchmark can be seen on figure 4.4.

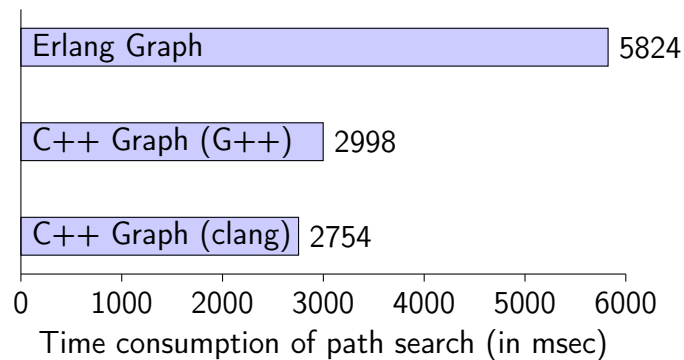


Figure 4.4: Performance comparison of the demonstration applications

From the figure it is clear that the C++ graph performs much better (about 2x is the performance gain) compared to the Erlang digraph. By compiling the C++ source code with clang, the result is a slightly more efficient program. In the C++ demonstration application both the chosen Erlang C++ interface and the prototype implementation of Erlang types were also measured which sounds fair.

4.4 Connecting Erlang and C++

Erlang compiles to bytecode which is executed by the Erlang Virtual Machine, namely the BEAM emulator. This means no native code is executed

by the VM by default. In the case someone wants to call a library function which is written in C or C++ Erlang provides four different mechanisms to do so:

1. Port
2. Linked-in driver
3. C-node
4. Natively Implemented Function

In this section all four are presented which is necessary to have an overview if we want to choose the right one for our implementation. In addition I would like to have an interface-independent graph implementation to be able to try all of them to find the most efficient for our goals.

4.4.1 Ports

The most basic type of connection between Erlang and C/C++. In this setup the library function is executed by an external operating system process started by the Erlang VM. The two operating system processes are communicating via their standard streams (stdin and stdout) with each other [7]. A great advantage of this layout is that a failure in the port does not affect the state of the VM that can recover and restart the port. On the other hand the greatest disadvantage is the way of communication. Needless to say it is inefficient to use a Unix pipeline for exchanging data, which is limited to transfer data 5000 bytes per second while the receiver reads only 100 bytes per second [20]. This solution is not preferable for a high-speed data connection between Erlang and a C/C++ driver.

4.4.2 Linked-in Driver

Linked-in drivers are C/C++ port drivers dynamically linked into the emulator process. The Erlang documentation states that this is the fastest way of calling C/C++ code from Erlang [9]. Truly, the calling requires no context

switches as it is required for the port solution. On the other hand a great disadvantage is that a crash in the port brings down the virtual machine, so unsafe code is very dangerous (can cause memory leaks also). The linked-in driver is a shared object (Unix/Linux) or dll (windows).

4.4.3 Erl Interface

Both in the previous and this case the programmer needs to specify a protocol in which the emulator and the process can exchange data. The Elr Interface provides a better approach: the programmer can use Erlang's external binary format instead of inventing a custom protocol [5].

4.4.4 C Node

As its name predicts, it is an operating system process other than the Erlang VM that acts like it were an Erlang VM [6]. The program setup is much like in the case of external ports using Erl Interface. It is a good choice if the C/C++ program needs to be externally available in a distributed network for more than one Erlang nodes. By choosing this one must face the overhead of the Erlang node-to-node communication via TCP/IP sockets. Obviously this is also not suitable.

4.4.5 Natively Implemented Function (NIF)

This is the newest method and possibly the fastest but surely the easiest to connect Erlang and C/C++. NIFs are usual functions of an Erlang module but implemented in C/C++ [10]. The calls are transparent to the caller process. Like linked-in drivers, this is also the fastest way of connection since it requires no context switches. The Erlang side implementation of the functions must exist as a fallback. By its nature the calls are synchronous in the process's point of view, and also atomic (the emulator cannot see what is inside the driver). Also the driver is linked into the VM, unsafe code can cause memory leaks and a failure can bring the whole emulator down. The main difference between the linked-in driver and NIF is that NIF is not tied

to a process, it is tied to an Erlang module (beam). This is an advantage in a multithreading environment if we do want to permit simultaneous access to the NIFs for not only one process.

4.4.6 Chosen Setup

In this thesis the Natively Implemented Function (NIF) will be chosen for its natural simplicity and additionally because this is the only one which permits accessing the internal representation of Erlang expressions directly inside the VM. As a long term goal the coming implementation shall support the replacement of the Erlang C++ interface to be able to try the others as well.

Chapter 5

Abstract Semantic Graph Representation

The chapter describes the main concept and details of the coming implementation. First the main requirements of the graph are subjected, with checking the possibilities and choosing an efficient representation. At the end of the chapter the new structure of the graph database will be presented.

5.1 Generic Graph Representation

It is necessary to find an efficient representation of the abstract semantic graph defined in Chapter 2. The main goal is to find a data structure which is optimized for the special operations, for example the *path* and *schema*. The most important is to define the requirements first, and based on these the actual data structures can be defined. These structures are wrapped by graph operations and also these are exposed through an interface towards the Erlang VM.

5.1.1 Concepts

The main requirements can be defined based on the abstract model of a node in the graph. The abstract model consist of pure data plus operations provided by the graph. The picture 5.1 helps us visualizing it.

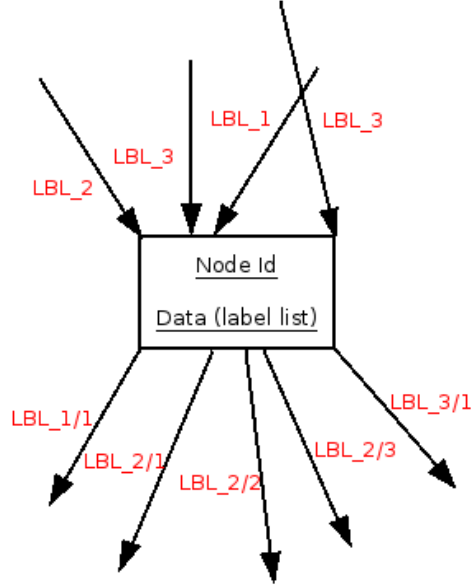


Figure 5.1: Abstract node

The node has a unique identifier and a list of labels as arbitrary data stored in the vertex. The data structure of a node could be a key and value pair, where the key is the unique identifier of the node, while the value is the data part ¹:

$$\begin{aligned}
 Node &= \{NodeId, Data\} \\
 Data &= \{Class, Label_1, \dots, Label_n\} \\
 NodeId &= nodeid_type() \\
 Class &= Label \\
 Label &= label_type()
 \end{aligned}$$

The *Class* is a special label. This identifies the class of the node. The arrows with *LBL_1*, *LBL_2* and *LBL_3* are edge labels defined in the graph schema. The number after the slash denotes the ordering between them. On the terminating edges no ordering can be defined, but it is very important

¹The syntax of type and data structure notation used is similar to the one used for Erlang

to have the forward ones ordered. The lexical layer of the model is a linked list of lexical nodes. The syntactic layer wraps around this list similarly like arbitrary Erlang expressions have their syntactic structure - an expression can contain subexpressions. Thus the order how the subexpressions are connected to a parent expression cannot be changed, so it is crucial to maintain the order of the links, moreover the exact copy of the original source code is needed to be reproduced by the tool on demand. This way it is necessary to store not only the list of forward links by labels, but to keep accounting the order of them. The data structure of links originating from a node can be decomposed to the following levels:

1. node id where the links originate
2. label of the links
3. index of the links, starting at 1 and unique within the node + label

These levels predict a not ordinary incidence list: the list of edges are ordered by index and grouped by the label. The data structure that represent link data:

$$\begin{aligned}
Links &= \{NodeId, TaggedLinks\} \\
TaggedLinks &= [\{Tag_1, Links_1\}, \dots, \{Tag_n, Links_n\}] \\
Links &= [\{Index_1, NodeId_1\}, \dots, \{Index_n, NodeId_n\}] \\
NodeId &= nodeid_type() \\
Tag &= label_type() \\
Index &= integer() > 0
\end{aligned}$$

In the above description the `[]` signs denote a collection which maintains its ordering and `{}` signs denote ordered pairs. The `*_type` are arbitrary data. To gain speed in graph traversals (section 2.7) the backward links are also stored per node. The reason is that when we want to move backwards on an edge in the graph we would have to traverse the graph towards the specified

node from the root point as a breadth-first search and if we found that node, then we are able to do the backward step. We can gain much speed if we store the backward edges as redundant data. Semantic layer does not need to preserve order on forward edges, but necessarily the graph does not need to distinguish between its layers, additionally the result of a semantic query will be received in the same order every time executed. The primary operations defined on the graph are strongly connected to the modeling and the utilization of the nodes. These are:

1. *create*: creates a new node in the graph, without any connections
 - data (arg): the data of the node as a tuple. The first element of the tuple is the class of the node.
 - returns: the id of the new node.
2. *delete*: deletes a node with any forward and backward links
 - id (arg): the id of the node to delete
 - returns: the atom 'ok'
3. *update*: updates the data part of a node
 - id (arg): the id of the node to update
 - data (arg): the new data to save
 - returns: the atom 'ok' or '{error, Reason}' if node does not exist.
4. *data*: returns the data contained by a node
 - id (arg): the id of the node to query
 - returns: the data of the node as a tuple or '{error, Reason}' if node does not exist.
5. *mlink* - creates a new directed link between two nodes with a specified link tag
 - from node (arg): the node where the link originates

- tag (arg): the label of the link, also this argument can be a tuple as a '{tag, position}' pair to specify the exact location where to insert the new edge.
- to node (arg): the node where the link terminates
- returns: the atom 'ok' or '{error, Reason}' if either one of the nodes does not exist or if the link is not allowed by the schema

6. *rmlink*: removes a link between the nodes

- from node id (arg): the node where the link to remove originates
- tag (arg): the label of the link
- to node id (arg): the node where the link to remove terminates
- returns: the atom 'ok' or '{error, Reason}' if the link is not allowed by the schema.

7. *links*: returns all forward links starting from a node

- node id (arg): the node the links are queried
- returns: '{ok, {Tag, [NodeId]}}', the node list is ordered by link index

8. *index*: returns the index of a specific forward link starting from a node

- from node id (arg): the node where the link originates
- tag (arg): the label of the link
- to node id (arg): the node where the link terminates
- returns: the index of the link

From this list it is obvious that the semantic analyzers expect the forward links preserving the order of edges. The *create*, *update*, *delete*, *data* are seem to be common database queries, so those can be propagated towards the database. The *mklink*, *rmlink*, *links* and *index* needs to be executed in an upper layer since these contain multiple database queries. While the first four operation are performed directly on the database, these four are on an

upper abstraction layer. Moreover, the latter four can be constructed by a sequence of the first four by including a fifth operation - *exists*:

- *exists*: predicate function returning true if data with the specified key exists.
 - data id (arg): the ID of data to be checked
 - returns: 'true' if data presents 'false' otherwise

This function is not part of the public API of the graph. Now with knowing the definition of *exists* now it is possible to define the remaining graph operations:

- *mklink* and *rmlink*
 - check if the two nodes exist
 - get the list of forward links of originating node
 - add / remove the specified link
 - store the list of forward links of originating node
 - get the list of backward links of terminating node
 - add / remove the specified link
 - store the list of backward links of terminating node
- *links* and *index*
 - check if the node exist
 - get the list of forward links
 -

To summarize, the *create*, *update*, *delete*, *data* and *exists* are related to a key-value store database, while *mklink*, *rmlink*, *links* and *index* are related to a graph. Graph nodes cannot be managed just by the key-value store operations because of the restrictions defined by the graph schema in 5.2.1. Instead, the extra actions are required to be done on an upper layer, connected to the

schema handling. All operations shall be implemented as visitor objects (or visitor functions) to separate data structures and algorithms. It gives us more flexibility in the implementation. These operations can be imagined as functions which operate directly on the data structures.

5.1.2 Graph Libraries

The previous subsection described the basic representation. Although both the Boost Graph Library (BGL) [2] and LEMON Graph Library (LEMON) [16] has the option to order edges in a graph, neither of them is prepared to insert links at a specific location. Moreover at graph traversal the order of links are also important. One could argue that the ordering works for example if we would use the *setS* as the edge iterator in BGL, but it's just by coincidence, it's not guaranteed by the library. In LEMON we do not know how the *EdgeIt* or the *ArcIt* was implemented. Additionally even if we would know that order is preserved on forward edges in LEMON, this behavior is not documented. However all operations could be implemented using BGL except the *mlink* (because of the edge position specifier), or using LEMON without the link management. The implementation needed for the abstract semantic graph is rather application specific that makes impossible to represent it using these libraries.

5.2 Abstraction From The Representation

The section provides a generic approach of graph algorithms based on the article Generic Graph Algorithms [26]. The main purpose is to be able to write algorithms which can work on mostly any kind of graph representation. Besides this it also makes possible to create higher abstraction layers to provide more efficient data access. This can be carried out by using two principles:

- Introducing an indirection between the representation and the data associated with nodes and edges.

- Algorithms has the main focus: instead of implementing them for one specific representation, they are built on top of the abstraction.

A trivial approach is to create converter classes that can convert data from the specific type used by algorithms to the types used in the representation and vice versa, requiring the conversion to be done every time data is accessed. This is likely to be inefficient: in the conversion important information may be lost about the graph structure, also these algorithms may perform in sublinear time. Instead, using a common denominator seems to be better: data of the representation can be accessed by iterators and data accessor objects depicted on figure 5.2.

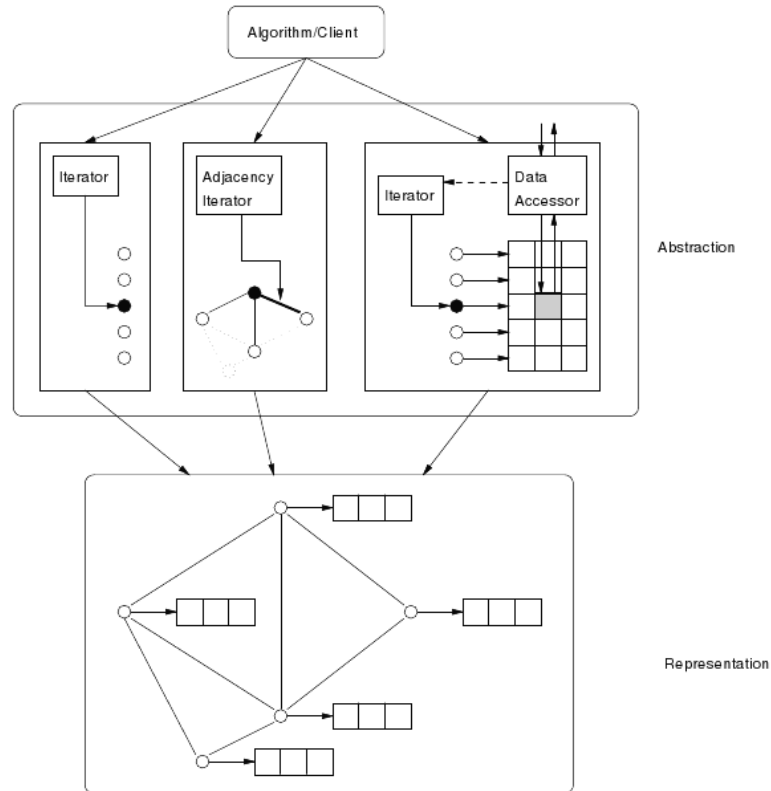


Figure 5.2: Concept of generic graph

An adjacency iterator is used to iterate through the adjacent nodes in a graph. It is required for the path function to traverse the graph in a BFS manner. Similarly a set of nodes may be accessed by a node iterator. Data

accessor objects know how to connect the iterator to the actual data and can provide a uniform interface towards the graph operations. The data accessor should be the component that provides the necessary basic functions, such as the previously described *create*, *update*, *delete*, *data* and *exits* since these are common for each piece of data we want to store. Accessing data associated with a graph node could be done by providing an iterator for the list of data labels we want to access. Similarly, the list of forward and backward edges are provided by edge label iterators, by them accessing edge iterators is possible to iterate through the specific edges which has the same label. The adjacency iterator is a special one. Using this the traversal of the graph is made possible: either by going through all edges of a specified node or by selecting a label first. The path function in an iteration step has to traverse the edges with a particular label and finally apply a *filter* function on the resulting set of nodes. In contrast, the *index* function requires the access to all edges from the node. The abstraction can provide both methods. Five iterators are needed:

- Node iterator
- Node data iterator
- Edge label iterator
- Edge iterator
- Adjacency iterator

In this idea data accessor is an abstract class, and concrete data accessors are need to be implemented per storage. It follows that because graph operations are not tied to the representation anymore, all operations except the basic five and algorithms can be implemented as function objects conforming the *visitor* design pattern [22]. Using these techniques a full separation of graph representation and algorithms can be achieved.

5.2.1 Graph Schema

The schema classifies the type of graph nodes and also defines the allowed type of edges between them. A node class is the type of the node, and the edge specification is the type of the edge. The schema creates the types of the graph elements. The schema is not static but is received dynamically as the first transaction after startup. This makes impossible to create the type classes beforehand and compile them for later use. Instead a reasonable way would be to create generic types which receive the type information run time when the object is being created. If a graph node is about to come into existence, the first thing to do should be to identify its type. This is a straightforward task, because the *create* method's argument - which is a tuple of single data elements - has the type identifier as the first element of the data tuple. But edges (links) are handled in another way. The type of edge is identified by the second parameter of the link manager function, namely the tag. This cannot be generalized as well, therefore the managing of nodes and edges need to be handled separately. There are necessary run time constraint checks based on the schema:

1. On creating a graph node, check if that class is defined in the schema
2. On creating an edge, check if the class of edge is defined by the schema
3. On executing a path expression, check if the next step in the graph is defined by an edge in the schema

The formal definition of the set of schema functions ($\mathcal{SF}$) are given in the following, based on the graph schema:

$$\begin{aligned}\mathcal{SCH} &= (C, N_c, A_c, E_c) \\ \mathcal{SF} &= (SC, SEC, SA, E_c),\end{aligned}$$

where

- $SC : C \rightarrow \text{boolean}$
- $SEC : C \times T \times C \rightarrow \text{boolean}$
- $SA : C \times A_n \rightarrow A_v$

The schema is strongly tied to a database, since it defines data types and relations between these. The best practice would be to define a thin layer between the storage and the graph operations where data is checked to conform the schema. If the check against the schema was successful then the action can be propagated to the key-value store. While there are several XML schema to source code generation projects for C++, all of them are based on compile time code generation, this way - unfortunately - these cannot be used: the graph is designed to receive the schema at startup so there is no way to know it compile-time. Optionally a relational database could be used here, where the tables and connections between them could be created based on the schema. For a relational database, there is no need to do extra constraint checks since those are already included in the database. Therefore two different schema handling layer should be created: one for key-value stores which does constraint checks and calls the appropriate key-value store action, and the other for relational databases which delegates these checks to the database. To provide the required abstraction, data accessors shall be the clients of the schema checks.

5.2.2 Path Function

The path function is the algorithm which provides efficient graph traversal. The application layers are using this to obtain data from the graph. It takes a node and a *path expression* as its starting arguments and traverses the graph in a breadth-first search manner. The *path expression* is a well-defined data structure, and acts as a query language. An important purpose of this thesis is to find an efficient implementation of the path function. The expressions that need to be processed are coming from Erlang as arbitrary data therefore interpretation is needed to be able to do the graph traversals fully in C++. The interpreter and the method of mapping Erlang data to C++

data is described afterwards in 5.3 and 5.4. It would be reasonable to create a generic path function because the schema can be changed, also the expression elements can be extended anytime if needed. Another reason to be generic is that *path expressions* are mainly based on the schema: the tag is an edge label defined in the possible node connections E_c . Graph traversal is need to be executed on an upper layer because it is a higher level algorithm. More details on the path function is defined in the next section because it is strongly dependent on the handling of *path expressions*, what is implicitly the low-level graph query language. In the next section we'll define a mapping of expressions to function objects. The formal definition of the path function:

$$\mathcal{SG} = (N, A_N, A_V, A, T, E),$$

$$\mathcal{PF} : NS \times F_T \times F_f \rightarrow NS,$$

where

- $NS \in N$
- $F_T : NS^{(NS \times T)}$
- $F_f : NS^{(NS \times f)}$
- $f = \mathcal{E}$

The notation B^A means a set of functions mapping from A to B, for example:

$$\varphi_T \in F_T \Rightarrow \varphi_T : NS \times T \rightarrow NS$$

and respectively

$$\phi_f \in F_f \Rightarrow \phi_f : NS \times f \rightarrow NS$$

The path function is designed to be a higher-order function object taking two function object parameters to be executed along with the set of nodes to operate on in each iteration step. Each element of the path expression can be processed separately predicting an iterative algorithm which handles one expression element at once. This kind of representation of the path function

and expressions provides a possibility of improvement. We can create an in-memory cache of the target set of nodes from frequently used queries or even for sub-queries. Also in-memory symbolic links can be created for frequently-used graph traversals providing a direct access to the resulting nodes.

5.3 Interpreting Path Expressions

An interpreter function is required to preprocess *path expressions*. These let the path function be able to execute it. The elements of the expression can be mapped to a corresponding *move* and *filter* function. A step in the graph is a function which - when executed - iterates through the current position's set of nodes and moves along the edges labeled by the given tag towards the specified direction. Similarly the filters can be functions which iterate through the list of nodes after the step in the graph and filters them. The formal definition of the interpreter:

$$\mathcal{M} : \mathcal{P} \rightarrow \mathcal{F}$$

$$\mathcal{M}(t_i, d_i, f_i) = \begin{cases} \varphi_i(t_i, F) & \text{if } d_i = \emptyset \text{ and } f_i = \emptyset \\ \varphi_i(t_i, d_i) & \text{if } f_i = \emptyset \\ \varphi_i(t_i, d_i) \circ \delta_i(f_i) & \text{else} \end{cases}$$

The function accomplishing the filter $\delta(f_i)$ has the formal type $\mathcal{E}$. The $\varphi(t_i)$ is the functions carrying out the one step move on the graph. Hence one path expression is mapped to one or two functions. For simplicity the *TrueFilter* is introduced to have the mapping result straightforward: it can be built up by a sequence of function pairs. These two functions can be passed to the path function in each iteration step. The mapping algorithm can be modeled as a deterministic finite automaton (DFA) [28]. As a path expression is mapped into two separate functions, for simplicity two DFAs are created: the graph moving function is depicted on figure 5.3 and the filtering is on figure 5.4. On the figures the $e()$ is a selector function which returns the specified member of the tuple.

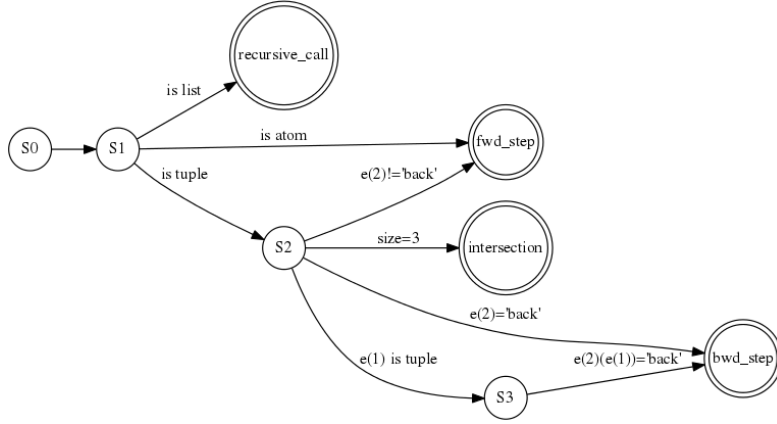


Figure 5.3: Interpreting graph moves

Graph move mapping Figure 5.3 depicts the DFA created to interpret graph move expressions. An arbitrary element of the *path expression* can be a nested list inside the expression. This means a recursive path expression, a list of edge labels to walk on, therefore a recursive call of the path function is needed. The result of the recursive call and the current state's node set are composing a union. If the element is an edge label (tag), it means a forward moving step in the graph. If the expression is a tuple and the second element is the atom called 'back' means a backward moving step in the graph. Also if the second element is not 'back' means a forward step. Moreover if first element of the tuple is also a tuple results in a backward step. If the size of the tuple is 3 means an intersection of the current state's node set and the sub-query result node set defined by the third element of the tuple. This is the only exception, since the intersection need to be mapped to a filter function, not to a step function.

Graph filter mapping Figure 5.4 depicts the DFA of the filter expression interpreter. First if the path expression element is not a tuple, it means that no filtering is defined. Here the TrueFilter will be applied, which can be implemented as a skip command. If it is a tuple, the elements of it needs to be investigated, importantly the second element. If it is the atom 'last', the LastFilter can select the last node of the set (Note that the set of nodes

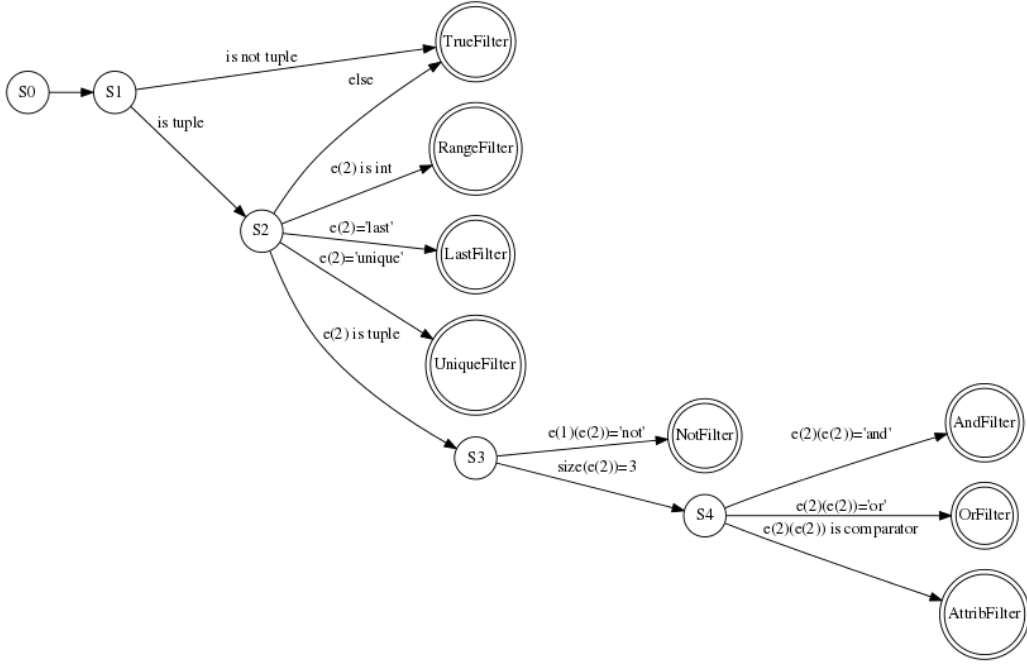


Figure 5.4: Interpreting graph filters

are ordered by the order of the forward links). This element can be an atom 'unique' meaning that the unique elements of the resulting set are kept. If it is an integer n , the n th element of the node set is kept. If it is a tuple, this tuple needs further investigation, else the result is also a TrueFilter. The first element of the new tuple is a 'not' atom means a NotFilter negating the effect of the supplied filter at position two of the new tuple. If the size of the new tuple is 3, the second element is checked: if it is the atom 'and', it results in the AndFilter, if it is the atom 'or', it results in the OrFilter and if it is a relation operator then it means the AttribFilter.

It is important to mention that although both the step and the majority of the filters need only the set of nodes at the current position as parameter, but the *intersection* filter makes it necessary to pass also the set of nodes of the previous step position along with the current position's. This way the functions are needed to be arranged into pairs, first element is the stepping function and the second is the filter. The path expression can be mapped into a sequence of function pairs. The path function will execute this sequence.

Moreover, if a function returns an empty set, it is inefficient to continue the algorithm because the result of any action on an empty set will be an empty set, the function shall return. To be more efficient, the expressions could be evaluated in a lazy way, to avoid unnecessary interpretations. A fully object-oriented approach is to create a *PathInterpreter* class which takes the expression at object creation and the stepping and filtering functions can be queried from it iteratively to be able to pass them to the path function in each iterative step. this class should implement the described DFAs. The interpreter is dependent on the current state of the query language so it needs to be separated from other, generic components.

5.4 Erlang Types

For efficiency it is necessary to implement the path function in C++. For storing arbitrary data, an internal data representation similar to the Erlang type model shall be created in C++. This is required by the interpreter. To handle pure data coming from Erlang and to store it in an internal representation in the graph, the study the Erlang type system is needed in order to make at least the minimal implementation of it in C++.

5.4.1 Type System

Erlang is a dynamically typed language which means that the type of data to operate on is not known at compile time, but deduced run time. A piece of data in Erlang is called a Term, which is an arbitrary Erlang expression. Term is a general 'title', it can be of any type. This is a main difference since C++ is statically typed, which means that the type of all expressions and subexpressions is known at compile time. Both of them has their advantages and disadvantages, but this thesis is not intended to describe these in details (see [25] for more), just to highlight the key differences between them. These are the minimum requirements for creating the model of Erlang's dynamic type system in the chosen statically typed language, C++.

Composite types in Erlang can be formed by using recursive data types

while in C++ composite types can be created as user-defined classes. Moreover in Erlang it is possible to compare data to each other regardless of the type of them. In C++ only the similar types of data can be compared (number, string and object) by the built-in relational operators. These may invoke implicit type casts as well, however these can be overloaded. From this point of view, the key differences are:

- Data representation: The type is known / not known at compile time. To avoid the extra type checks provided by C++ we need to introduce a general type for all Erlang types. See 5.4.2.
- Type deduction: The types are statically known / not known and dynamically deduced at run time. In C++ subtype polymorphism can be used and the dynamic type of data can be hidden run time behind a static general type. See 5.4.3.
- Recursion: Arbitrary Erlang data may have a recursive data structure, for example a list of numbers and strings may also contain sub-lists of data recursively. Primary C++ types are not recursive, but STL containers may contain recursively structured data but these have different semantics. The best solution is to use self-referring streams described in [24]. The topic is described in details in 5.4.4.
- Data comparison: in Erlang it is possible to compare any kind of data to each other regardless of the type of them. For example one may compare a list of integers with an atom which is a valid comparison. See 5.4.5.

It is important to mention that for arithmetic operators, the typing rules are more strict: it generates a run time exception if these are applied to incompatible types. For example, adding a string to a number: “string” + 4.

5.4.2 Erlang Term

First of all arbitrary Erlang data needs to be represented in a dynamic way, since Erlang types are evaluated run time. As in Erlang we call our data

a *Term* without knowing the exact type and data is hidden inside a *Term*. This concept makes straightforward to represent arbitrary Erlang data in an object oriented approach: a *Term* is an abstract type and the concrete types are derived from the *Term* class. It seems to be a good choice because it fits the object-oriented methodology: the static type of data will always be *Term*, while the dynamic type is the concrete one. This way the static type check will allow the code to be compiled. Additionally, the type classes can encapsulate all operations (such as the relational operators) making the *Term* to be an abstract data type. A wrapper type is also defined to provide an automatic memory management of *Terms*: the abstract *Term* is wrapped into a `boost::shared_ptr` naming the type *TermPtr*. A shared pointer is needed because a *Term* can be an element of another *Term* forming a data collection. If the pointer goes out of scope is not necessarily means that its lifetime is ended, it may live further inside this container. This way the references pointing to the concrete data element is need to be counted.

Private implementation Alternatively to provide more flexibility the class hierarchy can be extended to separate the concepts of *Term* and the representation of Erlang types having a special subtyping relation between them: they may have the same public interface so the concrete Erlang types are subtypes of *Term*, but not explicitly derived from it. This way we could use the *Term* as a wrapper of arbitrary Erlang data using the *Private Implementation* or in short, the *pimpl* technique: the concrete implementation is hidden inside an encapsulating class which has a pointer to the concrete implementation. The *Term* as a *pimpl* class could have a pointer to an abstract data type which is the super type of all concrete Erlang types while the *pimpl* may expose the public interface of the encapsulated data as it were its own. Moreover it also ensures no memory leak will happen since if the *Term* goes out of scope, its destructor deallocates the memory consumed by the concrete type ². Another advantage is that in our C++ graph we are not allowed to

²Because a term can be put recursively to a container it is necessary to count references to the concrete data before allowing the memory deallocation. It shall be permitted only if the reference count is reached 0

change data, only to store it without any transformation. The *pimpl* can also ensure data immutability by overriding the assignment operator. This way value can be assigned to a *Term* only once. This approach seems to be better since it shares the same semantics as the *Term* in Erlang originally.

5.4.3 Type Introspection

C++ does not have general and fully featured API for run time type introspection such as reflection in Java. It is necessary to create helper functions to provide the API which can be used to obtain run time type information. All concrete Erlang types need to have a constant and unique identifier, the *tid*, which can be checked run time. This identifier is strictly internal, but with the help of API functions the type can be easily determined at run time. Similar functions exist in Erlang, such as the *is_** BIFs (*is_number*, *is_atom*, *is_list*, ...) yielding true if the dynamic type of data matches. Similar ones shall be created. Also the usage of such functions are reasonable because these hide the actual data representation from the algorithms operating on them. Although the NIF interface has such functions, those are not preferable since we want to store data independent from the VM, so we need to be able to manage the memory allocation of data. This also leads us to the need of the internal type representation described here.

5.4.4 Recursive Data Structure

If data is received through the NIF interface, we will get an *ERL_NIF_TERM* array. The NIF library in Erlang has API which can be used to query the concrete type of the data just received from the VM. Using this we can build a term converter function which converts Erlang internal data to our newly created representation and vice versa. This converter needs to be called recursively if data has a recursive structure. The representation of these structures can be modeled as an Erlang container which is abstract, and the list and tuple are the descendants of the container. The elements of the container are also *Terms*. API functions are needed here to provide efficient access to elements of these structures. Such functions also exist in Erlang like the *ele-*

ment, *hd*, etc. These functions also hide the actual representation from higher level algorithms.

5.4.5 Type Precedence

When Erlang compares terms of different data types it does not do type conversion.³ Instead, to achieve dynamically typed data comparison, Erlang types have precedence classes. Comparing a data having a higher precedence class against another which have a lower, the higher precedence will have a higher value. The type classes in order of precedence:

1. Binary type
2. List type
3. Tuple type
4. Pid (process identifier) type
5. Port type
6. Function type
7. Reference type
8. Atom type
9. Numeric types, both integers and floating point numbers

The comparison of two arbitrary terms works as comparing the type classes first and if those are the same (equal precedence class), the comparison of the two values takes place. If we assign a descending integer number to each type class then we are able to compare any arbitrary data. This number can be the `tid`, the unique identifier of the concrete type introduced in the subsection 5.4.3. The object-oriented approach is to overload the relational operators (and also arithmetic if needed) of the type classes. Using this technique we can solve the data comparison problem.

³There is only one exception: comparing an integer and a floating point number using the `'=='` operator, the term having the lesser precision will be converted into the other term's type. Using the `':=='` or `'=/='`, no type conversion is done

5.4.6 Formal Definition of the Type System

Common functions

$$term = eval(expr)$$

All “term” expressions evaluate to a specific type:

$$\tau \in \{int, float, atom, reference, function, port, pid, tuple, list, binary\}$$

$$\mathcal{TH} = (type, float, class)$$

$$type : term \rightarrow \tau$$

$$float : term \rightarrow float, \text{ if } term \in \{int, float\}$$

$$class : term \rightarrow C_\tau, \text{ where } C_\tau \text{ is the type precedence class}$$

Helper functions for composite types

$$\mathcal{CTH} = (head, tail, size, elem)$$

$$head : list \rightarrow term_f, \text{ where } term_f \text{ is the first element}$$

$$tail : list \rightarrow list', \text{ } list' \text{ is the same as } list \text{ except the first element is removed}$$

$$size : tuple \rightarrow int, \text{ the element size of the tuple}$$

$$elem : int \times tuple \rightarrow term_n, \text{ where } term_n \text{ is the } N\text{th element}$$

Equation and comparison

$op \in \{<, \leq, >, \geq, =, \neq\}$

$$\mathcal{CMP}(t_1, t_2, op) = \begin{cases} \mathcal{CMP}_{list}(t_1, t_2, op) & \text{if } type(t_1) = type(t_2) = list \\ \mathcal{CMP}_{tuple}(t_1, t_2, op, 1) & \text{if } type(t_1) = type(t_2) = tuple \\ t_1 \text{ op } t_2 & \text{if } type(t_1) = type(t_2) \\ float(t_1) \text{ op } float(t_2) & \text{if } type(t_1) \in \{int, float\} \text{ and } \\ & type(t_2) \in \{int, float\} \\ class(t_1) \text{ op } class(t_2) & \text{else} \end{cases}$$

$$\mathcal{CMP}_{list}(t_1, t_2, op) = \begin{cases} head(t_1) \text{ op } nil & \text{if } t_2 = \emptyset \text{ and } t_1 \neq \emptyset \\ nil \text{ op } head(t_2) & \text{if } t_1 = \emptyset \text{ and } t_2 \neq \emptyset \\ nil \text{ op } nil & \text{if } t_1 = t_2 = \emptyset \\ \mathcal{CMP}_{list}(tail(t_1), tail(t_2), op) & \text{if } head(t_1) = head(t_2) \text{ and } \\ & t_1 \neq \emptyset \text{ and } t_2 \neq \emptyset \\ head(t_1) \text{ op } head(t_2) & \text{else} \end{cases}$$

$$\mathcal{CMP}_{tuple}(t_1, t_2, op, n) = \begin{cases} size(t_1) \text{ op } size(t_2) & \text{if } size(t_1) \neq size(t_2) \\ size(t_1) \text{ op } size(t_2) & \text{if } size(t_1) = size(t_2) = 0 \\ \mathcal{CMP}_{tuple}(t_1, t_2, op, n+1) & \text{if } elem(n, t_1) = elem(n, t_2) \\ & \text{and} \\ & size(t_1) > n \\ elem(n, t_1) \text{ op } elem(n, t_2) & \text{else} \end{cases}$$

5.4.7 Type System Object Model

In this subsection the object model is provided in the untyped lambda calculus. The following definitions and notation are based on [23]. The syntax of this calculus follows:

$$e ::= x \mid c \mid \lambda x. e \mid e_1 e_2 \mid \langle \rangle \mid e \Leftarrow m \mid \langle e_1 \Leftarrow+ m = e_2 \rangle \mid \langle e_1 \Leftarrow m = e_2 \rangle$$

In the grammar x is a variable, c is a constant symbol, $\lambda x.e$ is the lambda abstraction and $e_1 e_2$ is the application. The object forms follow:

- $\langle \rangle$ is the empty object
- $e \Leftarrow m$ is a message sending to e
- $\langle e_1 \leftarrow+ m = e_2 \rangle$ extends object e_1 with the new method m having the body e_2
- $\langle e_1 \leftarrow m = e_2 \rangle$ replaces the method m to have the body e_2 on the object e_1

Based on the above notation, the starting class type definition

$$\begin{aligned}
 term \stackrel{def}{=} & \langle \text{get_tid} = \lambda self.self \Leftarrow tid \\
 & cmp = \lambda self.\lambda x.\lambda op.null \\
 & (operator >) = \lambda self.\lambda x.cmp(x >) \\
 & (operator \geq) = \lambda self.\lambda x.cmp(x \geq) \\
 & (operator <) = \lambda self.\lambda x.cmp(x <) \\
 & (operator \leq) = \lambda self.\lambda x.cmp(x \leq) \\
 & (operator =) = \lambda self.\lambda x.cmp(x =) \\
 & (operator \neq) = \lambda self.\lambda x.cmp(x \neq) \\
 & \rangle
 \end{aligned}$$

The term type is incomplete, since it does not define its type id, the “tid” and the relational operators are abstract because of cmp , but in the subclasses these are completed. First, the type ids (tids) are represented by the following constants:

$$\begin{aligned}
 C_{tid} \in \{INT, FLOAT, ATOM, REFERENCE, FUNCTION, \\
 PORT, PID, TUPLE, LIST, BINARY\}
 \end{aligned}$$

The numeric types are defined in the following:

$$\begin{aligned}
typed_term &\stackrel{def}{=} \langle \quad term \\
&\quad \leftarrow+ val = \lambda self.0 \\
&\quad \leftarrow+ construct = \lambda x.\langle typed_term \leftarrow val = x \rangle \\
&\quad \rangle \\
is_same_type &\stackrel{def}{=} \lambda x.\lambda y.(== (x \Leftarrow get_tid)(y \Leftarrow get_tid)) \\
is_convertible &\stackrel{def}{=} \lambda x.(or(== (x \Leftarrow get_tid) INT)(== (x \Leftarrow get_tid) FLOAT)) \\
number &\stackrel{def}{=} \langle \quad typed_term \\
&\quad \leftarrow+ tofloat = \lambda self.(float (self \Leftarrow val)) \\
&\quad \leftarrow cmp = \lambda self.\lambda x.\lambda op.if is_same_type(self, x) then \\
&\quad \quad (op(self \Leftarrow val)(x \Leftarrow val)) \\
&\quad \quad else if (and(self \Leftarrow is_convertible)(x \Leftarrow is_convertible)) then \\
&\quad \quad \quad (op(self \Leftarrow tofloat)(x \Leftarrow tofloat)) \\
&\quad \quad \quad else (op(self \Leftarrow get_tid)(x \Leftarrow get_tid)) \\
&\quad \rangle \\
int &\stackrel{def}{=} \langle number \leftarrow+ tid = INT \rangle \\
float &\stackrel{def}{=} \langle number \leftarrow+ tid = FLOAT \rangle
\end{aligned}$$

The other “simple” types have the following definition. The atom and binary is provided here, since the others (function, reference, pid, port) are usable only for the VM, these cannot be represented externally. Please note that the string is special, it is represented as a list of ascii characters in Erlang language. However for simplicity, string is treated as a simple type in the

next subsection.

$$\begin{aligned}
simple_term &\stackrel{def}{=} \langle \quad typed_term \\
&\quad \leftarrow cmp = \lambda self.\lambda x.\lambda op. if \ is_same_type(self, x) \ then \\
&\quad \quad (op(self \Leftarrow val)(x \Leftarrow val)) \\
&\quad \quad else \ (op(self \Leftarrow get_tid)(x \Leftarrow get_tid)) \\
&\quad \rangle \\
atom &\stackrel{def}{=} \langle \quad simple_term \leftarrow+ tid = ATOM \rangle \\
binary &\stackrel{def}{=} \langle \quad simple_term \leftarrow+ tid = BINARY \rangle
\end{aligned}$$

In the following the composite types are defined:

$$\begin{aligned}
term_container &\stackrel{def}{=} \langle \quad typed_term \\
&\quad \leftarrow+ at = \lambda self.\lambda x.self_x \\
&\quad \leftarrow+ size = \lambda self.0 \\
&\quad \leftarrow+ append = \lambda self.\lambda x.self \leftarrow+ size = (self \Leftarrow size) + 1 \\
&\quad \leftarrow+ cmp_elems = \lambda self.\lambda x.\lambda y.\lambda op. \\
&\quad \quad if (and(== (self \Leftarrow at(y))(x \Leftarrow at(y))) \\
&\quad \quad \quad (and(> (self \Leftarrow size) 0)(> (x \Leftarrow size) 0))) then \\
&\quad \quad \quad (self \Leftarrow cmp_elems(x (+ y 1) op)) \\
&\quad \quad else (op (self \Leftarrow at(y)) (x \Leftarrow at(y))) \\
&\quad \rangle \\
tuple &\stackrel{def}{=} \langle \quad term_container \\
&\quad \leftarrow cmp = \lambda self.\lambda x.\lambda op.if is_same_type(self, x) then \\
&\quad \quad if (== (== (self \Leftarrow size)(x \Leftarrow size)) 0) then 0 op 0 \\
&\quad \quad elseif (== (self \Leftarrow size)(x \Leftarrow size)) then \\
&\quad \quad \quad (self \Leftarrow cmp_elems(x 0 op)) \\
&\quad \quad else(op(self \Leftarrow size)(x \Leftarrow size)) \\
&\quad \quad else(op(self \Leftarrow get_tid)(x \Leftarrow get_tid)) \\
&\quad \rangle \\
list &\stackrel{def}{=} \langle \quad term_container \\
&\quad \leftarrow cmp = \lambda self.\lambda x.\lambda op.if is_same_type(self, x) then \\
&\quad \quad if(and(== self nil)(/= x nil)) then nil op head(x) \\
&\quad \quad if(and(/= self nil)(== x nil)) then head(self) op nil \\
&\quad \quad else (self \Leftarrow cmp_elems(x 0 op)) \\
&\quad \quad else (op(self \Leftarrow get_tid)(x \Leftarrow get_tid)) \\
&\quad \rangle
\end{aligned}$$

5.4.8 Erlang Type Representation

The main class hierarchy of the created prototype implementation for Erlang type representation is described on the figure 5.5

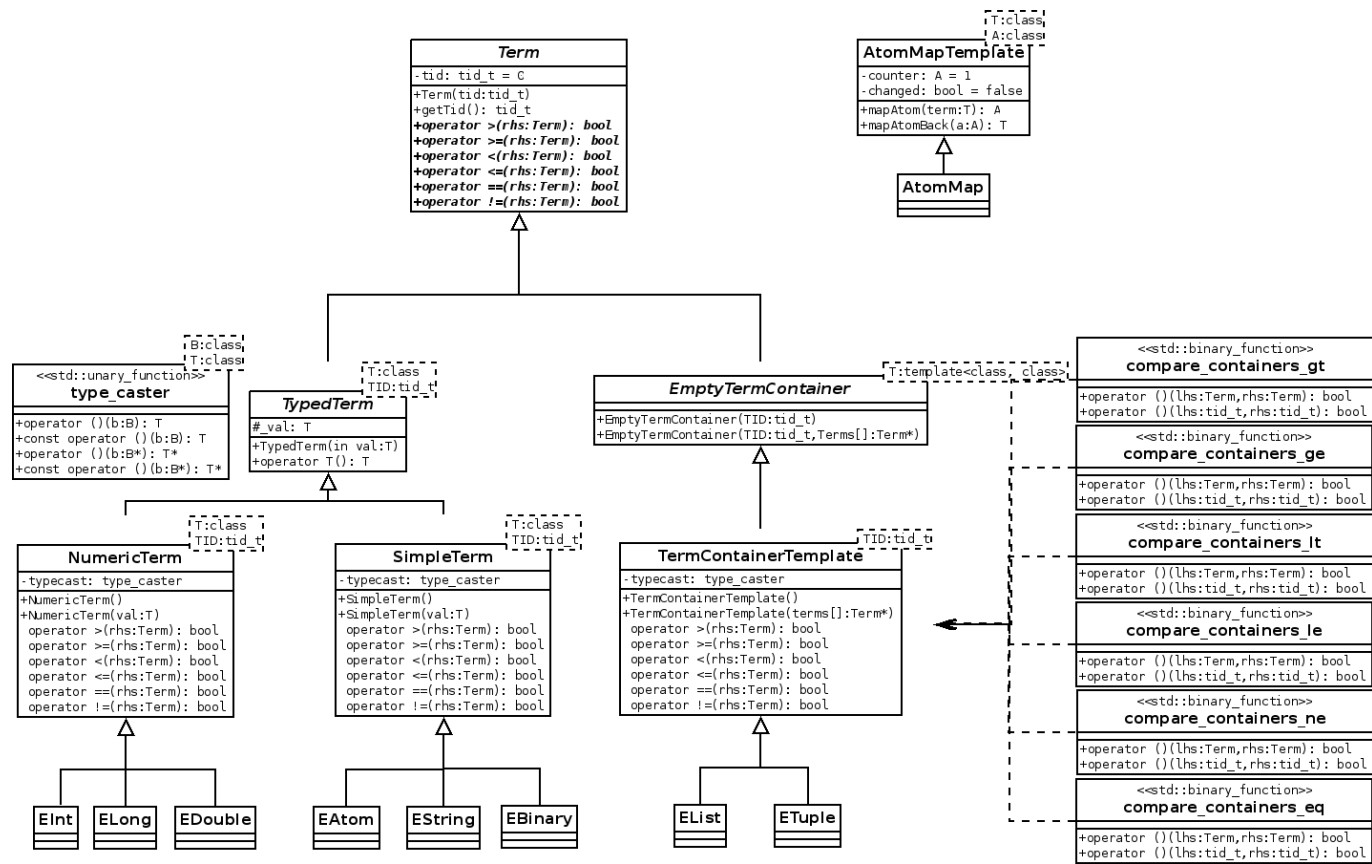


Figure 5.5: Erlang types in C++

The Term is an abstract class which is a base of the concrete types. Numeric types are an exception because we will need to compare integers also with floating point numbers. Some Erlang types are referring to internal data in the VM, but besides the identifier the actual data cannot be accessed at all. These data types are omitted here - *reference*, *function*, *port* and *pid*. These are dynamic types which represent a unique identifier which is assigned internally by the VM and their lifetime ends when the VM stops. It makes no sense storing persistently such data (for example for a function) which if the VM restarts, another identifier is assigned to. Furthermore we cannot access the bytecode represented by this identifier. A prototype implementation already exists and may be extended with the above described *pimpl* technique.

5.5 Abstraction Layers

In the previous sections of this chapter the graph representation was described as an abstract data type. The structure of the graph is monolithic, it consist of dependent component. It is important to define the layers to be able to create an implementation which is adaptive to changes. We shall define components as the implementation of the different layers. We shall make these components to be loosely coupled by allowing to be parameterized compile time. First look at the different layers on figure 5.6.

There is one part, the cache layer which is not described yet, but the section 6.7 will go into the details. The actual representation along with the storage model is the lowest one. At the same level there are the defragmenter and optionally the serializer (section 6.5) component. The schema-storage interface provides the abstraction needed to use both relational databases and key-value storages. The cache component is also defined here. The schema is the entry point of all basic data operations by providing the constraint checking facilities. Above these the graph operations and algorithms take place, using the Erlang data model created. The most upper layer is the Erlang/C++ interface itself providing the connection between Erlang and C++. The layers not described yet are defined in the following parts of the

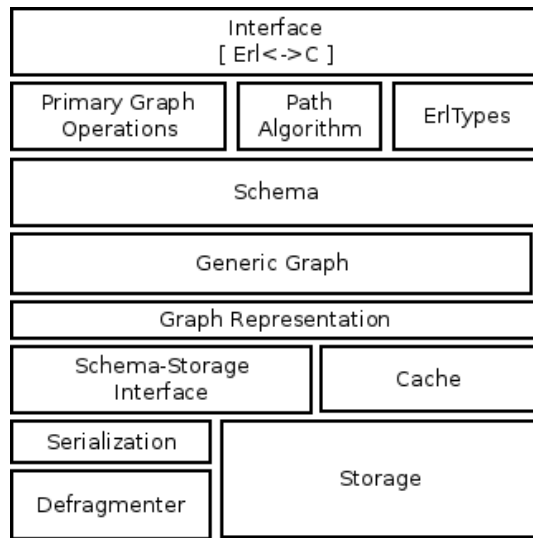


Figure 5.6: Abstraction layers

thesis, it was necessary to give the general concept here to provide better understanding on the following.

Chapter 6

Storage Model

6.1 Persistence Models

There are two storage concepts in general: the key-value store and the relational database. The thesis does not detail relational databases since in the first version of the tool an SQL based relational database was used and performed poorly. However the planned structure of the graph allows the usage of such data storages. The key-value store can be chosen from 2 alternatives: the first is based on Kyoto Cabinet 6.4, the second is a custom implementation based on the *mmap* [17] kernel API, the memory mapped file concept 6.2.

6.2 mmap

With *mmap* an ordinary file can be mapped into memory. The kernel treats this file as it were part of the virtual address space of the process and managing it as memory pages. This way it is possible to request the kernel to store data that copied there persistently in a file and that is also mapped into memory at the same time like it were a memory cache. The figure 6.1 shows the data flow in the mmap case.

Converting Erlang data to the appropriate C++ type representation, it is created directly in the space of the mapped file. The size of the file must



Figure 6.1: Data-flow with using mmap directly

be the multiple of the page size used by the operating system. Moreover the space allocation of the single pages are need to be accounted. Maintaining a mapping of the record identifiers and the pointers where the records are stored is necessary by putting these into a map data structure. Additionally the memory space needs to be accounted by a bitmap to be able to identify free space in memory and not to copy data eventually into the space of another record. The memory map needs to be maintained and stored per memory page, and its important that no data can flip over to the other memory page. These maps also need to be stored in the file persistently. Doing this requires to override the *new* operator: space occupying is done by setting the required number of bits in the memory occupation bitmap which are sufficient and returning the pointer of the starting address. We can use the place that fits first compared to the object size. Also by putting and removing data contained by nodes and links results in fragmentation. After a short time of usage, data will be scattered in the whole file. Even the elements of a record can be put sparsely into the file since these are created here first when the conversion from Erlang to C++ data happens. It is necessary to maintain some kind of order in data. Alternatively to save time on the implementation, the `boost::interprocess` library [3] can be used, fortunately it also supports multithreading, which has an important role in our implementation (section 6.8).

6.3 Defragmentation

If mmap is used to store all data of the graph it will suffer fragmentation. C++ is not a managed language, it does not have a garbage collector as Erlang has, so this task is onto us to complete. It is necessary to rearrange data in the mapped file time over time. To be specific, accounting every

bit of the space if occupied or not is not feasible. Thus the space needs to be divided into fixed size chunks. The idea how data is stored is a map of identifiers and a list of pointers pointing to the chunks occupied by the record. In defragmentation the whole record element needs to be rearranged and put into a new position. This can be done by first moving the container object and copying its member items one-by-one into the new position sequentially. The rearranging algorithm chosen is the first-fit with heuristics [12]. The first-fit algorithm works as copying data to the first coherent free space that is big enough. This algorithm connected to the bin-packing problem and has a time complexity of $O(n \log n)$ where n is the number of records to be packed. With the heuristic of considering the large items first will result in a $3/2$ approximation for bin packing, resulting the time complexity of $O(n^2)$. The defragmentation needs to be done after every fixed number of graph transformations.

6.4 Kyoto Cabinet

The figure 6.2 describes the data flow with Kyoto Cabinet.

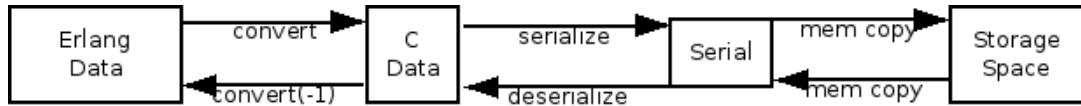


Figure 6.2: Data-flow with Kyoto Cabinet

Here at first arbitrary Erlang data is mapped into a C++ type representation. With these, data objects can be formed as graph nodes or edges. To store these in a Kyoto Cabinet storage a serialization is needed to put every piece of the data object into a single byte stream. This stream can be put into the storage rather than the individual data elements. The same is applied to the backwards direction: data returned by the storage needs deserialization to recreate its C++ type representation. This can be converted back into the original Erlang data. The complexity of implementation is much less by using Kyoto Cabinet. Next we will see the definition of the serialization method.

6.5 Data serialization

Thinking in an object-oriented approach this task can be delegated to the data objects, so every object should know how to serialize / deserialize itself. Additionally the recreation of the data structure has to be feasible from a pure byte stream. This is straightforward if we know at compile time what type of data will be stored in the stream. By serializing arbitrary data we cannot predict the types and values encoded in the stream. Thus we need to identify the type of data chunks. The already introduced type id - used in the Erlang type system model - perfectly fits here to be the type identifier in the stream. So the byte stream shall contain the following per data object:

1. *type identifier*
2. *size* optionally the size of data chunks if its a container type
3. *value* variable size of data chunk

The size of data chunk is not relevant if we assume that every type is a class and all of the classes know how to serialize / deserialize themselves. Additionally an external *serializer* object is needed which is able to produce a *serial* (a byte stream) from a data structure by calling the *serialize* method of it and passing itself as the function argument, and also able to do the deserialization of the data structure from a *serial*. This could be done by reading the type identifier first, creating that type of object denoted by the id and calling the *deserialize* method on the newly created object. Because data can be recursive - such as a data container, the *serial* must be embedded into the *serializer* object and need to be passed along the *serialize* and *deserialize* function calls. This way if the newly created object is a container, it could call the *serializer* to reproduce the elements of the container recursively. The *serializer* object will need to know the mapping between data structures and type identifiers.

6.6 Comparing The Persistence Models

The key differences are the following:

- Kyoto Cabinet needs data record creation by serialization
- Kyoto Cabinet - internally - does defragmentation
- mmap needs to do defragmentation
- mmap no data record creation is needed

While mmap saves the serialization process and the memcpy call, it needs more time on defragmentation, since not only the records need to be re-aligned, but the record elements. On the other hand, Kyoto Cabinet needs data record creation by the process of serialization. For performance reasons mmap seems to be a better solution, since there we can exploit the application specific usage of memory, for example we can use first-fit place of the free space. On the other hand while with Kyoto Cabinet the whole memory management and persistence is done, mmap needs to be handled carefully. This is better in the terms of safety, but not necessarily in performance.

6.7 In-memory Caching

This section describes an upper-level caching. The semantic analyzers are running parallel parsing and / or refactoring source code. Often there are a short sequence of database queries which are addressing the same query operation. This way there is a high change that the same data is requested by multiple analyzers, or even by the same analyzer. Another assumption can be made is that analyzers often operate on the same graph segment and the same set of nodes and edges at once. For example a function name rewrite transformation will start with the node of the function definition - *fundef* - and follow the *funcall* edges towards the set of nodes which represent the function call in an Erlang expression. If we assume that operations which are close to each other may contain parts of the same queries, with an in-memory cache data requests can be served from a higher level reducing the time needed for data retrieval. By using either kind of storage, Kyoto Cabinet or *mmap*, a caching is already done by the operating system (Kyoto Cabinet also uses *mmap*). Considering that both storages will produce page faults meaning

that some data being accessed goes out of the cache, it is reasonable to build an upper level in-memory cache for fast data retrieval. Another aspect of the caching is to shift some actions in time making it persistent. For example creating two new nodes are not necessarily need to become persistent until an edge is created between them. The cache layer other than the one provided by the storage could do this. Also a persistence checking can be done at this level by identifying orphaned nodes having no edges when a user operation is finished on the graph. All orphaned nodes predict an error in the semantic analyzers. For the first implementation a FIFO would be created to provide caching with fixed number of records. Later the possibility of implementing a least recently used list is reasonable since graph operations are not sequential, but rather random access. This component of the graph database needs to be separated from the storage to provide a transparent layer for upper-level components.

6.8 Multithreading

Multithreading has an important role, because Erlang from the R12B release can run multithreaded. First the concept of parallel execution in Erlang is presented, explaining its role in RefactorErl and a better approach to congestion handling is introduced.

6.8.1 Multithreading in Erlang

In the early versions, there was only one processor core and one thread executing the bytecode - also input/output operations had a separate thread in Erlang to provide asynchronous I/O. In contrast, parallel execution is achieved by creating cheap individual processes inside the virtual machine that can run parallelly. This means that the VM has an internal *scheduler* which has the task of utilizing processor time between the processes need to run. There is the *run queue* which is a FIFO containing all processes which need to execute bytecode. The scheduler allows one process to run at once. After a certain amount of reductions done, it interrupts the current process

and puts it back to the end of the *run queue*. On a multi-core CPU, the Erlang VM creates one execution thread per core by default. All execution threads has an own scheduler and run queue. This means that if multiple analyzer processes running at the same time in RefactorErl, these will be put into separate run queues, and thus the database will be accessed parallelly. The tool is already prepared for it with a readers-writer lock.

6.8.2 Readers-writer Lock

The lock is implemented on the graph level. This means that all data fetching query can run parallelly, but if a transformation query is received by the graph, no other query is started and the transformation query received by the graph is also halted until the data fetchers finish. After there are no more queries running, the transforming operation is executed individually. Here we do not know if the data fetchers are querying the same data as the transformer is about to change. In most of the time it can be assumed that there is no overlapping between the readers and the writer. We can say that this is not efficient. The aim is to implement a more optimal lock by moving the locking mechanism from the graph to the node level. It can be assumed that this will result in much less congestion. This approach leaves only one question: what happens if the data to be accessed does not exist yet but about to be created. It can happen that two analyzer processes want to create the same node in the graph. Because this does not exist yet, two nodes of the same could be created. Fortunately this can be avoided by an upper level component used by the analyzers, the *nodesync*. This prevents creating the same nodes more than once. By manipulating edges, we can avoid congestion of creating them twice by storing the empty edges records when the graph node is created. This was we can assume that by moving the readers-writer lock to the node and edge record level is safe. With all these it can be assumed that by moving the readers-writer lock to a lower level will provide less congestion and saves idle execution time on multi-core CPUs.

Chapter 7

Conclusion

The thesis showed the formal definition of the abstract semantic graph which represents Erlang source code. Prototype implementations are extended, database abstraction is done by me, also existing databases are described and compared by efficiency. Improvement areas are studied in order to find a better overall solution for performance problems on storing a graph in a database. The key concepts are the definition a more optimal representation, extending multithreading support, moving up the layer of the Erlang - C++ interface. Also a dynamic type model is created in C++ which behaves similarly as Erlang data. A new representation is introduced for the abstract semantic graph, and two new storage models are described as alternative implementations. Studying the topic of graph databases beginning from the formal graph representation used in RefactorErl was productive and data access optimization is generally achieved.

7.1 Future Improvements

There are areas of improvements, application-specific features which can be further analyzed:

- building an incidence matrix on-demand, after the finalize of user operation.
- find a better defragmentation algorithm

- find a better caching algorithm
- find a lightweight representation of the graph which could be loaded into memory and could improve performance of graph traversal.
- create symlinks between the most often used queries and target set of nodes as a fast in memory cache.

Bibliography

- [1] Benchmarking gcc vs clang. http://www.phoronix.com/scan.php?page=news_item&px=MT
- [2] Boost Graph Library. http://www.boost.org/doc/libs/1_52_0/libs/graph/doc/index.html.
- [3] Boost interprocess library. http://www.boost.org/doc/libs/1_52_0/doc/html/interprocess.html.
- [4] Clang is said to be better. <http://clang.llvm.org/comparison.html>.
- [5] Erl interface. http://www.erlang.org/doc/tutorial/erl_interface.html.
- [6] Erlang c node. <http://www.erlang.org/doc/tutorial/cnode.html>.
- [7] Erlang c port. http://www.erlang.org/doc/tutorial/c_port.html.
- [8] Erlang digraph library. <http://www.erlang.org/doc/man/digraph.html>.
- [9] Erlang linked-in driver. http://www.erlang.org/doc/tutorial/c_portdriver.html.
- [10] Erlang NIF. <http://www.erlang.org/doc/tutorial/nif.html>.
- [11] Erlang Programming Language. <http://erlang.org>.
- [12] First fit algorithm with heuristics. http://www2.informatik.hu-berlin.de/alkox/lehre/lvws1011/coalg/bin_packing.pdf.
- [13] Graph database. http://en.wikipedia.org/wiki/Graph_database.
- [14] Kyoto Cabinet. <http://fallabs.com/kyotocabinet/>.
- [15] Kyoto Cabinet kcdbopen() function reference.
http://fallabs.com/kyotocabinet/api/kclangc_8h.html#a464237fa818e4b70b15526a898b5c5

- [16] Lemon graph library. <http://lemon.cs.elte.hu/trac/lemon>.
- [17] mmap man page. <http://www.kernel.org/doc/man-pages/online/pages/man2/mmap.2.html>.
- [18] Mnesia DBM. http://www.erlang.org/doc/apps/mnesia/Mnesia_chap1.html.
- [19] RefactorErl Homepage. <http://plc.inf.elte.hu/erlang>.
- [20] Unix pipeline. [http://en.wikipedia.org/wiki/Pipeline_\(Unix\)#Implementation](http://en.wikipedia.org/wiki/Pipeline_(Unix)#Implementation).
- [21] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, Tejfel. M., and M. Tóth. Refactorerl - source code analysis and refactoring in erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, pages 138–148, Tallin, Estonia, October 2011.
- [22] Christopher G. Lasater. *Design Patterns*. Wordware Applications Library, 2006.
- [23] Kathleen Fisher, Computer Science Dept, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. In *Nordic Journal of Computing*, pages 26–38, 1994.
- [24] Attila Góbi, Zalán Szügyi, and Tamás Kozsik. A C++ pearl – self-referring streams. *Annales Univ. Sci. Budapest., Sect. Comp.*, pages 157–174, 2012.
- [25] Nyékné Gaizler J. *Programming Languages*. Kiskapu, 2003.
- [26] Dietmar Kühl. Generic graph algorithms. In Mehdi Jazayeri, Rüdiger G.K. Loos, and David R. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 249–255. Springer Berlin Heidelberg, 2000.
- [27] Books Llc, S. Wikipedia, and B. Group. *Hash Functions: Cryptographic Hash Functions, Cyclic Redundancy Check, Sha-1, Md5, Ripemd, Pearson Hashing, Collision, Random Oracle*. General Books, 2011.

- [28] R. Mak. *Writing Compilers and Interpreters: A Software Engineering Approach*. Wiley, 2011.
- [29] Lajos Rónyai, Gábor Ivanyos, and Réka Szabó. *Algoritmusok*. Typotex, 1999.
- [30] M. Tóth and I. Bozó. Static analysis of complex software systems implemented in erlang. Central European Functional Programming Summer School – Fourth Summer School, CEFPP 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743, 2012.

Appendix A

Graph Measurement Commands

The command used for measuring the time of loading Mnesia library.

```
timer:tc(ri, add, ["/usr/lib/erlang", mnesia]).
```

The method which was used to measure the query execution time.

```
A=[timer:tc(ri,q,["mods[name=mnesia_monitor].funs[name=get_env]
    .refs.param"]) || _ <- lists:seq(1,1000)].
lists:foldl(fun({X,_},Acc) -> Acc + X end, 0, A) / length(A).
```

The following is a fully detailed table about the measured values

Action	time (sec)	CPU (%)	MEM (MB)
<i>Mnesia db</i> , loading of mnesia	1615.984127	63	640
<i>Mnesia db</i> , query	0.151010	73	640
<i>Nif Graph</i> , loading of mnesia	240.140322	42	516
<i>Nif Graph</i> , query	0.060583	59	529
<i>Kyoto Cabinet</i> , loading of mnesia	352.519855	44	310
<i>Kyoto Cabinet</i> , query	0.093673	56	312

Table A.1: Measurement of prototype implementations

Appendix B

Source Code of Demonstration Applications

The Erlang source code is one module which is used for both applications, but compiled differently. The *cpp* compile-time macro is used to create the C++ version.

```
-module(graphtest).

-export([start/1,
        get_cycles_for_root/1,
        build_paths/1,
        load_nif/0,
        vertices/1,
        test/1,
        build_paths/2]).

-ifdef(cpp).
-define(INIT, digraph:new()).
-define(ADD_VERTEX(G, V, Label), digraph:add_vertex(G, V, Label)).
-define(ADD_EDGE(G, V1, V2, Label), digraph:add_edge(G, V1, V2, Label)).
-define(GET_PATH(G, Root, V), digraph:get_path(G, Root, V)).
-define(CHOOSE_ROOT(G), choose_root(G)).
-define(VERTICES(G), digraph:vertices(G)).
```

```

-define(DELETE(G), digraph:delete(G)).
-else.
-define(INIT, init_graph()).
-define(ADD_VERTEX(G, V, Label), add_node(V, Label)).
-define(ADD_EDGE(G, V1, V2, Label), add_edge(V1, V2, Label)).
-define(GET_PATH(G, Root, V), get_path(Root, V)).
-define(CHOOSE_ROOT(G), choose_root_cpp()).
-define(VERTICES(G), vertices()).
-define(DELETE(G), ok).
-endif.

-define(NIF_ERROR, throw(nif_not_loaded)).

load_nif() ->
    erlang:load_nif("./graphtest", []).

test(Ns) ->
    G = start("../graph.txt"),
    N = lists:seq(1, Ns),
    [spawn(?MODULE, build_paths, [self(), G]) || _ <- N],
    [P | _] = [receive P -> P end || _ <- N],
    ?DELETE(G),
    P.

start(File) ->
    case file:consult(File) of
        {ok, Terms} ->
            build_graph(?INIT, Terms);
        {error, Reason} ->
            erlang:error(Reason)
    end.

build_graph(G, [{vertex, V, Label} | Tail]) ->
    %digraph:add_vertex(G, V, Label),
    ?ADD_VERTEX(G, V, Label),

```

```

    build_graph(G, Tail);
build_graph(G, [{edge, V1, V2, Label} | Tail]) ->
    %digraph:add_edge(G, V1, V2, Label),
    ?ADD_EDGE(G, V1, V2, Label),
    build_graph(G, Tail);
build_graph(G, []) ->
    G.

get_cycles_for_root(G) ->
    [V | _] = digraph:vertices(G),
    digraph:get_cycle(G, V).

build_paths(Pid, G) ->
    Pid ! build_paths(G).

build_paths(G) ->
    %Root = choose_root(G),
    Root = ?CHOOSE_ROOT(G),
    build_paths(G, Root, ?VERTICES(G), [{Root, Root, []}]).

vertices(G) ->
    ?VERTICES(G).

% same vertex, skip
build_paths(G, Root, [Root | Vs], Paths) ->
    build_paths(G, Root, Vs, Paths);
build_paths(G, Root, [V | Vs], Paths) ->
    %NewPath = digraph:get_path(G, Root, V),
    NewPath = ?GET_PATH(G, Root, V),
    build_paths(G, Root, Vs, [{Root, V, NewPath} | Paths]);
build_paths(_, _, [], Paths) ->
    Paths.

choose_root(G) ->
    Vs = digraph:vertices(G),

```

```

        choose_root(G, Vs).

choose_root(G, [V | Vs]) ->
    case digraph:vertex(G, V) of
        {V, [root]} ->
            V;
        _ ->
            choose_root(G, Vs)
    end.

choose_root_cpp() ->
    choose_root_int().

%%% c funs

init_graph() ->
    ?NIF_ERROR.

add_node(_, _) ->
    ?NIF_ERROR.

add_edge(_, _, _) ->
    ?NIF_ERROR.

choose_root_int() ->
    ?NIF_ERROR.

get_path(_, _) ->
    ?NIF_ERROR.

vertices() ->
    ?NIF_ERROR.

```

The C++ source code:

```

#include <iostream>
#include <map>
#include <lemon/list_graph.h>
#include <lemon/bfs.h>
#include <lemon/path.h>
#include "test03/typeid.h"
#include "test03/erlang/types.hpp"
#include "test03/erlang/efuns.hpp"
#include "test03/erlang/nif/nif.hpp"
#include <erl_nif.h>

static lemon::ListDigraph* g;
// needed for arcs
static std::map<erlang::TermPtr, lemon::ListDigraph::Node>* nodes;
static std::map<lemon::ListDigraph::Node, erlang::TermPtr>* nodeIds;
static std::map<erlang::TermPtr, erlang::TermPtr>* labels;
static erlang::nif::NifTermDecoder* decoder;
static erlang::nif::NifTermEncoder* encoder;

typedef std::map<erlang::TermPtr, lemon::ListDigraph::Node>
    ::iterator NodeIt;

template<class InputIter, class Key>
static InputIter findByKey(InputIter begin, InputIter end, Key& k)
{
    InputIter it = begin;
    while (it != end)
    {
        if ((*it->first) == (*k))
        {
            return it;
        }
        ++it;
    }
}

```

```

    return end;
}

static ERL_NIF_TERM initGraph(ErlNifEnv* env, int argc,
    const ERL_NIF_TERM argv[])
{
    g = new lemon::ListDigraph();
    nodes =
        new std::map<erlang::TermPtr, lemon::ListDigraph::Node>();
    nodeIdIds =
        new std::map<lemon::ListDigraph::Node, erlang::TermPtr>();
    labels = new std::map<erlang::TermPtr, erlang::TermPtr>();
    decoder = new erlang::nif::NifTermDecoder();
    encoder = new erlang::nif::NifTermEncoder();
    return enif_make_atom(env, "ok");
}

static ERL_NIF_TERM addNode(ErlNifEnv* env, int argc,
    const ERL_NIF_TERM argv[])
{
    // just ignore the label now...
    erlang::TermPtr nodeId = decoder->decodeTerm(env, argv[0]);
    lemon::ListDigraph::Node n = g->addNode();
    //(*nodes)[nodeId] = n;
    nodes->insert(std::pair<erlang::TermPtr,
        lemon::ListDigraph::Node>(nodeId, n));
    (*nodeIdIds)[n] = nodeId;
    return enif_make_atom(env, "ok");
}

static ERL_NIF_TERM addEdge(ErlNifEnv* env, int argc,
    const ERL_NIF_TERM argv[])
{
    erlang::TermPtr fromNodeId = decoder->decodeTerm(env, argv[0]);
    erlang::TermPtr toNodeId = decoder->decodeTerm(env, argv[1]);

```

```

NodeIt ni1 = findByKey(nodes->begin(), nodes->end(), fromNodeId);
NodeIt ni2 = findByKey(nodes->begin(), nodes->end(), toNodeId);
if (ni1 != nodes->end() && ni2 != nodes->end())
{
    lemon::ListDigraph::Node n1 = ni1->second;
    lemon::ListDigraph::Node n2 = ni2->second;
    g->addArc(n1, n2);
    return enif_make_atom(env, "ok");
}
return encoder->encodeTerm(env, toNodeId);
}

static ERL_NIF_TERM chooseRootInt(ErlNifEnv* env, int argc,
    const ERL_NIF_TERM argv[])
{
    return encoder->encodeTerm(env, (*(nodeIds->begin()))>second);
}

static ERL_NIF_TERM getPath(ErlNifEnv* env, int argc,
    const ERL_NIF_TERM argv[])
{
    erlang::TermPtr rootId = decoder->decodeTerm(env, argv[0]);
    erlang::TermPtr nodeId = decoder->decodeTerm(env, argv[1]);
    erlang::TermPtr terms = E_NIL;
    NodeIt rooti = findByKey(nodes->begin(), nodes->end(), rootId);
    NodeIt nodei = findByKey(nodes->begin(), nodes->end(), nodeId);
    if (rooti != nodes->end() && nodei != nodes->end())
    {
        lemon::Bfs<lemon::ListDigraph> bfs(*g);
        bfs.run(rooti->second);
        if (bfs.reached(nodei->second)) {
            erlang::list(terms)->push_back(nodeId);
            lemon::ListDigraph::Node prev = bfs.predNode(nodei->second);
            while (prev != lemon::INVALID) {
                erlang::list(terms)->push_back((*nodeIds)[prev]);
            }
        }
    }
}

```

```

        prev = bfs.predNode(prev);
    }
}

erlang::TermPtr ePath = E_TUPLE(terms);
return encoder->encodeTerm(env, ePath);
}

static ERL_NIF_TERM vertices(ErlNifEnv* env, int argc,
    const ERL_NIF_TERM argv[])
{
    erlang::TermPtr list = E_NIL;

    std::for_each(nodes->begin(), nodes->end(),
        [&list](std::pair<erlang::TermPtr,
            lemon::ListDigraph::Node> pair)
        {
            erlang::list(list)->push_back(pair.first);
        });
    return encoder->encodeTerm(env, list);
}

static ErlNifFunc nif_funcs[] =
{
    {"init_graph", 0, initGraph},
    {"add_node", 2, addNode},
    {"add_edge", 3, addEdge},
    {"choose_root_int", 0, chooseRootInt},
    {"get_path", 2, getPath},
    {"vertices", 0, vertices},
};

ERL_NIF_INIT(graphtest, nif_funcs, NULL, NULL, NULL, NULL)

```

The source of the graph is defined here as Erlang tuples containing a type identifier as the atom 'vertex', a unique identifier and a label. For edges, the type identifier is followed with the source node id, the destination node id and a list of labels:

```
{vertex, 1, [root]}.  
{vertex, 2, [2]}.  
{vertex, 3, [3]}.  
{vertex, 4, [4]}.  
{vertex, 5, [5]}.  
{vertex, 6, [6]}.  
{vertex, 7, [7]}.  
{vertex, 8, [8]}.  
{vertex, 9, [9]}.  
{vertex, 10, [10]}.  
{vertex, 11, [11]}.  
{vertex, 12, [12]}.  
{vertex, 13, [13]}.  
{vertex, 14, [14]}.  
  
{edge, 1, 2, [1,2]}.  
{edge, 1, 3, [1,3]}.  
{edge, 1, 4, [1,4]}.  
{edge, 2, 5, [1,2]}.  
{edge, 2, 6, [1,2]}.  
{edge, 3, 7, [1,2]}.  
{edge, 4, 8, [1,2]}.  
{edge, 5, 9, [1,2]}.  
{edge, 6, 8, [1,2]}.  
{edge, 7, 8, [1,2]}.  
{edge, 9, 10, [1,2]}.  
{edge, 10, 11, [1,2]}.  
{edge, 11, 12, [1,2]}.  
{edge, 11, 7, [1,2]}.  
{edge, 11, 13, [1,2]}.  
{edge, 11, 14, [1,2]}.
```

```
{edge, 12, 13, [1,2]}.  
{edge, 12, 14, [1,2]}.  
{edge, 13, 2, [1,2]}.  
{edge, 13, 5, [1,2]}.  
{edge, 14, 2, [1,2]}.  
{edge, 14, 9, [1,2]}.
```

Appendix C

Commands Used for the Demonstration

The Erlang source code was compiled using

```
erlc graphtest.erl
```

The command for the *cpp* version:

```
erlc -Dcpp graphtest.erl
```

The C++ code was compiled as

```
g++ -O3 -shared -I /usr/lib/erlang/usr/include/  
-o graphtest.so -std=c++11 -lemon -fPIC cgraph.cpp  
clang++ -O3 -shared -I /usr/lib/erlang/usr/include/  
-o graphtest.so -std=c++11 -lemon -fPIC cgraph.cpp
```

The following functions are defined to automatically execute the graph test functions from the Erlang interpreter shell:

```
F = fun() -> graphtest:test(5) end.  
F2 = fun(0, Fun) -> F(); (N, Fun) -> F(), Fun(N-1, Fun) end.
```

The test command was the following:

```
timer:tc(F2, [100, F2]).
```