STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Special Issue 1, 2014
10th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, May 21-25, 2014

ERLANG-LIKE DYNAMIC TYPING IN C++

ANDRÁS NÉMETH AND MELINDA TÓTH

Abstract. Each programming language has its type system. In simple terms, when the type checking takes place at compile time we call this language statically typed. On the other hand, if the type checking is done at runtime, it is a dynamically typed language. In this paper we present the key differences between the two approaches by choosing two general-purpose programming languages, one from each category.

Erlang is a dynamically typed language with an infrastructure for developing highly available, distributed applications. However for computation-intensive tasks for performance reasons a statically typed language would be a better choice. This paper introduces a method of interoperability between Erlang and C++ in the perspective of type systems.

1. Introduction

Each general-purpose programming language has unique properties which make some more favourable compared to others in the respect of solving specific problems. We often write big programs using one general-purpose programming language that is good enough for the most of our intentions, but often might not be the best choice on a few but essential areas. To overcome such complications, we can piece our program together using elements written in different programming languages, resulting in a mixed construction that can be an optimal solution for that problem domain.

In this paper, we choose two programming languages: Erlang [9] and C++ [22]. On one hand, Erlang is a good choice to achieve greater productivity, easy cluster communication, or to write soft real-time applications. On the other hand, with C++, one can produce more efficient programs, since the emitted machine code is much more optimised and performs better than interpreted and platform-independent bytecode. Putting these together can

Received by the editors: May 1, 2014.

2010 *Mathematics Subject Classification.* 68N15, 68N19.

1998 *CR Categories and Descriptors.* D.3.3 [**Programming Languages**]: Language Constructs and Features – *Data types and structures*; F.3.1 [**Logics and Meanings of Programs**]: Semantics of Programming Languages – *Algebraic approaches to semantics*.

Key words and phrases. Connecting programming languages, Erlang, C++, interoperability, type system, data mining, static code analysis, graph representation.

be a good choice if the computation-intensive parts of a distributed application is written in C++ while the rest in Erlang. But there is one fundamental difference between the two: these have completely different type systems; while Erlang is dynamically typed, C++ is statically typed. In order to write parts of our program in C++, modelling the type system of Erlang is needed since we want to write algorithms in both that can operate on the same data. In favour of ensuring seamless transitions from one to the other and vice versa, we should be able to specify the type of data even when we do not know in advance what the data is. The main goal of this method is to write the computation-intensive parts of a program in C++ that operates on Erlang data and thus behave as it is in Erlang.

2. Motivation

RefactorErl [11, 6] is a source code analysis and transformation tool for Erlang. The tool provides several features to support program development and maintenance, for example refactorings, software metrics, code browser and source navigation by semantic queries or bad smell detection. RefactorErl analyses the Erlang sources in advance, builds a Semantic Program Graph [14] from it and stores the graph in a database. Later, when some information is required, it gathers the information from this database.

Implementing computation-intensive parts of a program in C++, and using it from Erlang aims to deliver faster data access along with more stable code comprehension capabilities of RefactorErl. It was observed that long queries can consume much time that can be unacceptable for the end user. For instance finding function references globally when several millions of functions are parsed would take up long minutes.

The Semantic Program Graph is a directed, labelled graph, and attributes are assigned to each type of vertex. It has one specific node which has no incoming edges, this is the *root node*. The types of vertices and the edge labels between them are described by the *graph schema*. This is not known at compile time, rather when the tool starts, semantic analysers register themselves and describe the data they want to store and attaching their own parts to the global graph schema. This way the tool does not know in advance what will be the type of data it has to store and operate on. There are already existing graph implementations in C/C++ for RefactorErl using already existing techniques to connect Erlang and C/C++ [3], but the handling of arbitrary data is not solved by those, rather these parts of the graph is still written in Erlang. Consequently to overly optimise the tool we need to start from the bottom first: to implement the whole Semantic Program Graph along with the traversal

algorithm in C++ and utilising hardware resources by executing platform-specific machine code.

The graph traversal algorithm is used to extract semantic information from the graph. A starting node (or set of nodes) is given and a graph path which is mainly a series of edge labels. One part of this path expression can contain a direction specifier and an additional filtering specification besides the edge label. These are optional parts, and hence the expression contains different types of elements. Therefore the exact type of the expression is not known in advance, additionally a path can contain another path recursively. When evaluating filters the processed data is influenced by the current graph schema.

3. Type Systems

Price [21] defined type systems in the following way: "A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute." He characterises its main purpose as to rule out certain program behaviours that are undefined or have unpredictable results. These are considered programming errors which cause runtime failures in future program executions. Most of these errors (behaviours) can be checked statically, but not all of them: the well-known division by zero, bad typecasts or null pointer dereferences are the most common examples. These mistakes cause rather runtime exceptions, therefore the static type check shall include these as valid results of the execution: the semantics of these expressions contain whether the current evaluation is valid or not [12]. Phrases can be classified: a type system can be imagined as a formal system composed of rules which assign a type to each language construct of a programming language.

A computer cannot distinguish an integer value from the executable code of a function, it just sees sequences of bits stored in the memory: it is an untyped universe, and operations are carried out on bit sequences. Therefore the type system is a fundamental aspect of language design [16]: each language determines its own way how it classifies bit sequences: a type gives the semantic meaning to a particular sequence of bits. These constraints are verified by the type checker given by the programming language. This checker identifies the language parts referring to a specific construct (e.g. variable identifier, function identifier, etc.) and checks if the values used in these expressions satisfy the type constraints required by that particular construct.

Type check can take place early at compilation time by the compiler. This method is called static type checking. It requires the programmer to annotate every identifier in the program with a type keyword depending on the kind of data it is intended to hold during execution. The types in this method does

not depend on the possible execution paths since the annotations already restrict the potential values a language construct can contain. If type checking succeeds, the program is considered type safe. For the time of program execution the type information is not kept since a type safe program has no more need to verify values during runtime. This results in faster program execution.

On the other hand, dynamic type checking takes place at the time when the program is executed. The execution environment does the type checking and type information is inferred based on the exact execution path currently being evaluated. The evaluation succeeds and execution continues if the type check succeeds, hence the execution path is type safe. If it fails, the program execution is stopped and an exception is thrown. On the other hand, it does not require the written program code to be annotated, rather the programmer can write code intuitively and often can be more productive. However it requires the programming errors to be foreseen.

A type checker can only perform semantic verification of program code on a relatively low level where the language defines the static meaning of its language constructs, therefore the analysis of algorithmic fulfilment takes place on a higher abstraction level. This requires other methodologies to prove program correctness that are out of scope of this paper.

4. Interoperability Between Type Systems

In the introduction the cooperation of programming languages is mentioned: writing program code in two different general-purpose programming languages and attaching them. In this paper these two chosen languages are Erlang and C++. For this combination interoperability is already available using several already existing methods [3]. However in these cases the successful communication requires the types of exchanged data to be consistent between the program parts participating in the cooperation. For our purpose 2 it is not sufficient, since there are cases when the type of data is unknown at compile time. To form the next level of interoperability, an agreement between the two languages shall be defined that concludes that data must be utilised the same way on both sides semantically. This includes the semantics of the fundamental computational operations such as relational, arithmetic and logical expressions. Type information is a key concept since it needs to be preserved at runtime to be able to extract it later if the program execution makes it necessary. This implies that at some point of the program it might be irrelevant what type of data is used. The semantics of a certain value is employed only when the evaluation of the current expression makes it unavoidable. Algorithms written in both languages leveraging on this agreement facilitate the seamless interoperability between them. On the type system level, language

constructs other than the previously mentioned ones such as control statements are not considered, because these are mainly related to program control and not to type systems [8]. We consider this part as a different topic and therefore should be approached differently.

4.1. **Key Concepts of the Erlang Type System.** In the type system of Erlang every language construct has a particular type which is inferred by the runtime system. In type theory terms are the elements of the set of base values. In any case, when an expression is evaluated its result will be a term in the Erlang terminology. The static type of this term needs to be preserved during program execution.

The particular type of a term is deduced runtime when a value is involved in the evaluation of an expression. Although it is dynamic, it does not allow automatic type conversions [1], therefore the type system is strict, and type information is hidden until that data is used in an expression. Type inference implies the ability to examine the static type of a piece of data at any stage of execution - this is type introspection - and attach the precise semantics of the current operation. Both Erlang and C++ uses eager evaluation, therefore it is sufficient to introspect the terms involved in the current subexpression, the whole expression is not checked for type safety beforehand. This implies that shortcut evaluation of logical expressions would succeed in some cases when the first operand satisfies the type constraints and the other is not. More importantly, Erlang uses single assignment. When a term is assigned to a variable for the first time, that value is associated with that identifier. Next time the assignment acts like a pattern matching, and succeeds if the right hand side expression evaluates to the same value that is stored in the variable on the left hand side. Otherwise a runtime exception is thrown indicating that the pattern matching is failed.

Recursive data structures can be used in Erlang. For example a list can contain other lists of arbitrary data recursively. Types in C++ cannot be recursive, since the size of a recursive type is infinite. Similar effect can be achieved using pointers or STL [7] containers. However these have different semantics since these use indirection. Although a recursive data structure can be implemented in C++ as self-referring streams [13], these are not sufficient for handling arbitrary data.

Erlang is very permissive in data comparison: any kind of data can be compared regardless of the type of data they actually store. For example a list of integers can be compared to a floating-point number which yields a boolean value (to be more specific, a term containing a boolean value). In

[1]The implicit conversion between floats and integers in an exemption

order to allow comparison of any kind of data, the ordering between types is defined:

$$Binary > List > Tuple > Pid > Port > Function > Reference > Atom > Numeric\ types$$

When the comparison of different types occur, the one having the higher ordering considered to be the greater *value* than the other.

When terms are involved in arithmetic or logical expressions Erlang is not this permissive. These expressions are defined between certain types having the same place in the ordering and a runtime type error is raised if this is not satisfied. For example adding an integer to a string raises a "badarith" or "badarg" exception during execution if the expression is arithmetic or logical respectively.

4.2. **Formal Definition of the Type System.** In this section the formal semantics of the the Erlang-like type system is written, based on the observations described in the previous section. Terms are the elements of the set of base values. All "terms" have a particular type:

$$T = \{int, float, atom, reference, function, port, pid, tuple, list, binary\}$$

T is a set of type categories where each element represents a type. This is the type information preserved in the runtime system. We have a typing function *type* for this purpose, that extracts type information from an arbitrary term at runtime.

$$type : term \to T$$

The helper functions are used to define the operations:

$class : term \to int$, where int is a constant integer number
$float : term \to float$, if $type(term) \in \{int, float\}$
$head : term^{list} \to term$
$tail : term^{list} \to term^{list}$
$length : term^{list} \to int$
$size : term^{tuple} \to int$
$elem : term^{int} \times term^{tuple} \to term$

The *class* function is used to classify terms. It yields the constant number that the terms are associated with based on their type category. These constants are numbers that are strictly monotone increasing as the type ordering is defined. *head*, *tail* and *length* are functions operating on lists. The first one returns the first element of a list, the second one returns the rest of the elements except the first of the given list, while *length* returns the number of

elements a list currently contains. The functions *size* and *elem* operate on tuples, *size* returns the number of elements a tuple contains and *elem* returns an element of it based on the given index. Term comparison is done in the following way:

$op_c \in \{<, \leq, >, \geq, =, \neq\}$
$bool = \{true, false\} \mid true, false \in atom$
$\mathcal{C} : term \times term \times op_c \to term^{bool}$
$\mathcal{C}_l : term \times term \times op_c \to term^{bool}$
$\mathcal{C}_t : term \times term \times op_c \times int \to term^{bool}$

All relational operations return a bool type. Although Erlang does not have such type, the atoms *true* and *false* are returned. Accordingly these two special values implicitly produce the set of boolean. This is referred to as bool type in the followings, although this is not a distinct type in Erlang. The notation of op_c refers to a comparison operator and $\mathcal{C}$, $\mathcal{C}_l$, $\mathcal{C}_t$ are total functions that carry out the comparison using the given terms and relational operator. The semantics of these functions follow:

$$\mathcal{C}(t_1, t_2, op_c) = \begin{cases} \mathcal{C}_l(t_1, t_2, op_c) & \text{if } type(t_1) = type(t_2) = list \\ \mathcal{C}_t(t_1, t_2, op_c, 1) & \text{if } type(t_1) = type(t_2) = tuple \\ t_1 \ op_c \ t_2 & \text{if } type(t_1) = type(t_2) \\ float(t_1) \ op_c \ float(t_2) & \text{if } type(t_1) \in \{int, float\} \ and \\ & \quad type(t_2) \in \{int, float\} \\ class(t_1) \ op_c \ class(t_2) & \text{otherwise} \end{cases}$$

The first step is to check the type of the operands involved in the comparison. If both are lists or tuples, then the list or tuple compare function carries out the operation, respectively. Otherwise if both of them has the same type, the stored values are compared. If floating point and integer numbers are compared, the integer is converted to a floating point value. The comparison without these implicit type casts are not considered here.

$length(nil) = 0$

$$\mathcal{C}_l(t_1, t_2, op_c) = \begin{cases} length(t_1) \ op_c \ length(t_2) & \text{if } t_1 = \emptyset \quad or \quad t_2 = \emptyset \\ \mathcal{C}_l(tail(t_1), tail(t_2), op_c) & \text{if } head(t_1) = head(t_2) \ and \\ & \quad t_1 \neq \emptyset \ and \ t_2 \neq \emptyset \\ \mathcal{C}(head(t_1), head(t_2), op_c) & \text{otherwise} \end{cases}$$

Nil is the list which has the length of 0. Comparing two lists starts with comparing the its elements one by one, taking the head first. When one of the lists has no more elements, it is considered having the less value and the other having the greater. If the elements are equal, the comparison continues taking the next list element. The last rule is applied when the current elements are not equal, this case the original comparison function has to be used.

$$\mathcal{C}_t(t_1, t_2, op_c, n) = \begin{cases} size(t_1) \quad op_c \quad size(t_2) & \text{if } size(t_1) \neq size(t_2) \text{ or} \\ & \quad size(t_1) = size(t_2) = 0 \\ \mathcal{C}_t(t_1, t_2, op_c, n+1) & \text{if } elem(n, t_1) = elem(n, t_2) \\ & \quad and \quad size(t_1) > n \\ \mathcal{C}(elem(n, t_1), elem(n, t_2), op_c) & \text{otherwise} \end{cases}$$

Tuples with different sizes are compared by comparing the number of elements they contain. If the size is the same, the elements are compared one by one. If the currently analysed elements are equal, the comparison continues with the next elements. Otherwise the original comparison function is used, this case the current values are equal. Arithmetic operations can be defined only on numeric types, and has the following signature:

$op_{ua} \in \{+, -\}$
$op_a \in \{+, -, *, /\}$
$op_b \in \{band, bor, bxor, bsl, bsr\}$

$\mathcal{A} : term \times op_{ua} \to term^{\tau} \cup \bot$, where $\tau \in \{int, float\}$
$\mathcal{A} : term \times term \times op_a \to term^{\tau} \cup \bot$, where $\tau \in \{int, float\}$
$\mathcal{A}_{rem} : term \times term \to term^{int} \cup \bot$
$\mathcal{A}_{bin} : term \times term \times op_{bin} \to term^{int} \cup \bot$

Above, the op_{ua} denotes a unary arithmetic operator + or −, op_a is a binary operator in the sense that it takes two arguments, and op_{bin} is a bitwise arithmetic operator. Integer divisions are not considered here, although Erlang has a special operator - *div* - for this purpose. $\mathcal{A}$ is a total function which takes two terms and an operator and evaluates the expression based on the type of operand(s). It has two special cases: *"rem"* and *"bin"* that need to be handled separately. The return type of this function is merely depend on the type of operands. If both are integers, the result type is *int*, else it is *float*. The only exception is the division which yields a floating point number in all cases. If the type of operands are not valid (other than *int* or *float*)

the operation fails. The same behaviour is expected when the $\mathcal{A}_{rem}$ or $\mathcal{A}_{bin}$ is used with terms having a type other than *int*. Failure means that the evaluation function raises a runtime exception. The evaluating rules of arithmetic operations are described here in the following:

$$\mathcal{A}(t, op_{ua}) = \begin{cases} op_{ua}\ t & \text{if } type(t) \in \{int, float\} \\ \\ \bot & \text{otherwise} \end{cases}$$

Unary arithmetic operations are defined on an int or float operand. In these cases the operation succeeds and otherwise fails and a runtime exception is generated.

$$\mathcal{A}(t_1, t_2, op_a) = \begin{cases} t_1\ op_a\ t_2 & \text{if } type(t_1) = type(t_2)\ and \\ & \quad type(t_1), type(t_2) \in \{int, float\} \\ \\ float(t_1)\ op_a\ float(t_2) & \text{if } type(t_1), type(t_2) \in \{int, float\} \\ \\ \bot & \text{otherwise} \end{cases}$$

In these binary arithmetic operations the types int and float are considered compatible. If both operands are either integers of floats, the operation succeeds and the result type will be integer of float respectively. If only one of the operands is an integer and the other is a float, an implicit type conversion occurs on the integer number, and the result type is float. The computation of the remainder of an integer division is a special case and defined only if both of the operands are integers:

$$\mathcal{A}_{rem}(t_1, t_2) = \begin{cases} t_1\ rem\ t_2 & \text{if } type(t_1), type(t_2) = int \\ \\ \bot & \text{otherwise} \end{cases}$$

Binary arithmetic operators follow: binary and, or, xor, shift left and shift right:

$$\mathcal{A}_{bin}(t_1, t_2, op_b) = \begin{cases} t_1\ op_b\ t_2 & \text{if } type(t_1), type(t_2) = int \\ \\ \bot & \text{otherwise} \end{cases}$$

These operations are defined only on integer operands else a runtime exception is generated.

Logical expressions are defined in the following:

$op_{ul} \in \{not\}$
$op_l \in \{not, orelse, andalso\}$
$\mathcal{L} : term \times term \times op_l \to term^{bool} \cup \bot \; \mathcal{L}_{not} : term \to term^{bool} \cup \bot$

The operation *not* negates the result of a logical expression or variable containing a logical value. It is a unary operator, therefore the $\mathcal{L}_{not}$ evaluates these expressions. The other two - *andalso* and *orelse* are shortcut logical operations since in C++ we have only one kind of infix operator for each logical operation, therefore having a regular *and* a shortcut *and* operator is not feasible. In C++, logical operators use shortcut evaluation, these seem more natural and therefore here only these are the ones considered. The other exception is that despite Erlang defines a logical *xor*, C++ lacks such an operator. The function $\mathcal{L}$ evaluates the logical operator op_l with the two term operands and returns a bool type term or fails if the operands does not have the type of bool. In case of shortcut evaluation a valid result is returned if only the first operand has the type of bool and the second operand need not to be evaluated.

$$\mathcal{L}(t_1, t_2, op_l) = \begin{cases} t_1 \; op_l \; t_2 & \text{if } type(t_1), type(t_2) = bool \\ \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{L}_{not}(t) = \begin{cases} not \; t & \text{if } type(t) = bool \\ \\ \bot & \text{otherwise} \end{cases}$$

The function $\mathcal{L}$ evaluates a binary logical expression. It expects bool type operands. Similarly, the unary $\mathcal{L}_{not}$ expects a bool type term.

5. Implementation Considerations

5.1. **Erlang Term.** The concrete type of Erlang data is checked at runtime. In the Erlang terminology, all element in the set of base values is called a term: it has an exact type, but it remains hidden until the value contained has to be extracted. This concept makes straightforward to represent arbitrary Erlang data object oriented: the term can be an abstract class that exposes all operations defined on a value (such as the relational operators). The concrete data types can be derived implementing all these operations. To provide more flexibility and runtime efficiency, the separation of the required interface and the concrete data types would be desirable. Another advantage is that automatic memory management can be done easily if the concrete data is

stored in a standard shared pointer inside the term [2]. This method is called a *private implementation*, or in short *pimpl*. In this case, there is no explicit subtyping relation between term and the concrete data. Both exposes the same interface, therefore this is a special polymorphic relation we call duck typing [23]. The third advantage is that data immutability can be ensured by overloading the assignment, copy assignment and move assignment operators. An empty term stores a piece of data when it is constructed or when a value is assigned to it for the first time. Since the assignment is fully controllable, at subsequent assignments the new value can be matched against the contained data if those are the same, and an exception can be generated if not.

5.2. **Type Introspection.** C++ does not have a general language feature that supports runtime type introspection, for example like reflection in Java. It is necessary to create predicate functions that can be used to obtain runtime type information and to hide the actual type tags: all concrete Erlang types need to have a constant unique identifier, the `tid`, which is checked runtime to determine the type of a term. Similar functions exist in Erlang, such as the `is_number`, `is_atom`, `is_list`, etc., yielding true if the static type of the argument matches. Also the usage of such functions are reasonable because these hide the actual data representation from the algorithms operating on them.

5.3. **Recursive Data Structures.** List and tuple types can be composed recursively (e.g a list containing additional lists). The concrete Erlang list and tuple classes can contain an implementation-specific container (e.g. an intrusive list, array, etc.) which contains instances of terms, just like in Erlang. The manipulation of container data cannot be done directly using member functions of the container, but with free functions provided by the implementation. Similar ones are also available in Erlang, like *element*, *hd*, etc.

5.4. **Type Ordering.** When Erlang compares terms of different data types it does not perform implicit type conversions [3] like scripting languages often do, for example when a string value is put into an arithmetic expression, it is implicitly converted to an integer or float, depending on the representation of the stored value. Instead, to achieve dynamically typed data comparison, different Erlang types are compared based on the ordering between them: data

[2]Because 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.

[3]There 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 type of the other term. Using the '=:=' or '=/=', no type conversion is done.

type on a higher place in the ordering against another which have a lower, is like comparing a higher number against a lower one: $number < atom < reference < function < port < pid < tuple < list < binary$. The number used for relational operations can be the same as the `tid` - the unique identifier introduced in the subsection 5.2. The object-oriented approach is to overload the relational operators (and also arithmetic) of the concrete types therefore the runtime behaviour of the current data is altered depending on its static type using dynamic dispatch.

5.5. **Prototype Implementation.** The main class hierarchy of the prototype implementation of the Erlang type system is depicted on Figure 1.

Term is the class that wraps a value as an instance of the *AbstractTerm*. This prevents illegal access to the representation of the encapsulated value, hence there is no convenient way to extract the data stored inside. Term is *Immutable*, it prevents re-binding data to an already assigned one by generating an exception. An operation on a term can be done only through *AbstractTerm*. It stores a type id that can be retrieved runtime to determine the type of data. This is an abstract class that does not implement the operations. This is the supertype of concrete data types: the *TypedTerm* which encapsulates a single value. This is a generic class and has no other role than to introduce and store typed values. *GuardedTerm* implements the abstract interface, and has a *Type* type parameter which determines what type of data will be stored in its *TypedTerm* supertype. *TypeId* is the constant number which can be used runtime for type determination. This value is stored in the *AbstractTerm* to allow operations to extract runtime type information without without the need of typecasting. It also has a *TermOperator* that carries out an actual operation - relational, arithmetic or logical - on the operands. A *TermOperatorGuard* is bound as a type parameter. This ensures that operations can only be performed on allowed types of data. Type checking and type exception generation takes place here. Some data types - *reference*, *function*, *port* and *pid* - are not shown here, because these represent values that are interpretable and usable only by the Erlang virtual machine. Although these could be stored here in a special binary format, these are considered superfluous in the processing of semantic data.

6. Related Work

First to mention, the type system of Erlang have mostly studied in the area of static program code analysis. Simon Marlow and Philip Wadler presented in the paper *A Practical Subtyping System For Erlang* [19] a type system for Erlang with the typing rules of Erlang expressions. They created a prototype type checker that was able to infer the types of expressions of static program

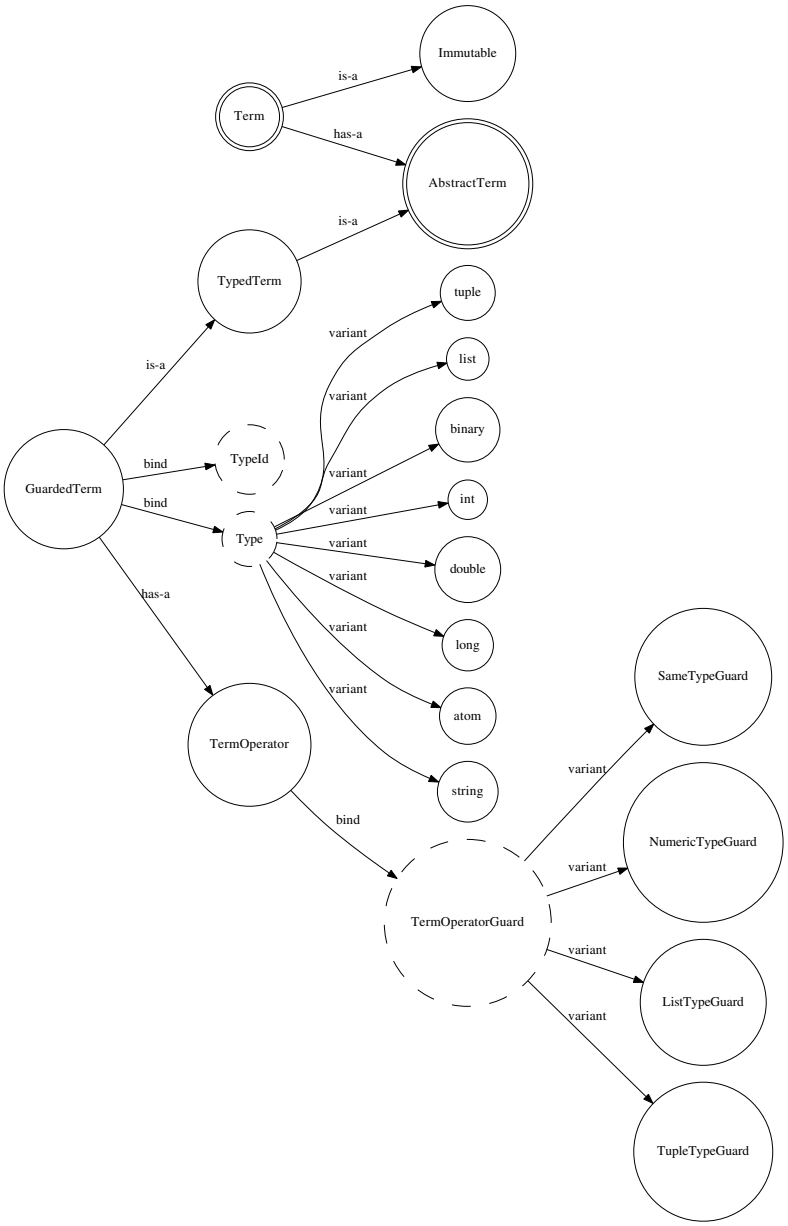

FIGURE 1. Erlang Types in C++

code. With similar goals, the work of Tobias Lindahl and Konstantinos Sagonas was aimed to inspect and verify the correctness of program code by creating

the TypeEr [18]. They have defined the basic types of the Erlang type system with the typing rules required to infer and check the types of expressions on possible execution paths based on the success typing [17]. Sven-Olof Nyström suggested a soft-typing system for Erlang [20]. This method introduced the checking of the source code of an Erlang module using data-flow analysis: it verifies the correctness of the module's interface described by a specification language. All of these three are focusing on the static type checking of program code within the boundaries of Erlang.

Connecting two programming languages often involves a high-level interface description language (IDL). The contract between programming languages can be described using this IDL. Zoltán Horváth, Zoltán Varga, Viktória Zsók wrote about this method in *Clean-CORBA Interface for Parallel Functional Programming on Clusters* [24]. This work introduced the IDL compiler which was able to translate the objects described in CORBA [2] to the language constructs and code stubs of Clean [1]. With this method, the Clean functional language could be interfaced with others.

The topic of type system interoperability for distributed applications was also studied by Valérie Issarny and Amel Bennaceur in the paper *Composing Distributed Systems: Overcoming the Interoperability Challenge* [15]. The paper describes various possible techniques of interoperability on the middleware layer including software components running distributively but not in a closely-related manner. This was based on the usage of ontologies. Similarly Gordon S. Blair, Massimo Paolucci, Paul Grace and Nikolaos Georgantas represented ontology-based interoperability in large systems in the paper *Interoperability in Complex Distributed Systems* [10]. Their work introduced a method of connecting different components of a system on a higher level, using semantic web services and semantic middleware.

7. Conclusions

A prototype implementation exists satisfying the theoretical background described in this paper. Our assumption said that the same algorithm runs more optimally in C++ than in Erlang if a similar type system is used. To confirm this, a measurement is done. Two programs were compared: the first is written fully in Erlang, using the digraph [4] library. All data is read from a text file and using this the graph was created in memory using digraph. Graph data contained the vertices, edges and a piece of data stored by each vertex. The measurement is started at this point: a shortest path search was done between the root node of the graph and every other node which existed. The second program was written in Erlang and C++. The graph and graph processing algorithm was in C++, using the LEMON [5] graph library.

All graph processing was made here, using the Erlang type system model to build the graph and to store the data there. The measurement involved the interface usage and data conversion overhead. A function is called in Erlang from where the control of execution is transferred to the C++ program. The results are shown on figure 2. Despite of the preprocessing overhead - the

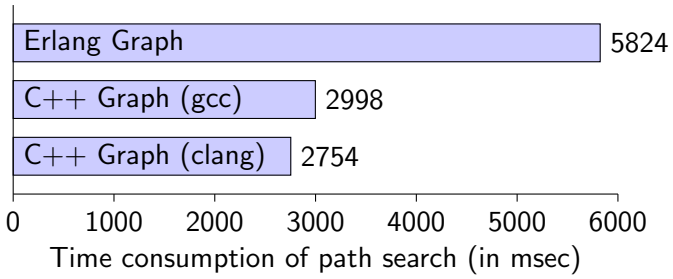

FIGURE 2. Performance Comparison of the Demonstration Applications

type conversion and language change - the C++ program run faster, although during the measurement just a small amount of data was processed.

In the first part of this paper, the observations of Erlang data types along with their behavioural attributes were presented. Afterwards a model was introduced that described the semantics of Erlang terms in relational, arithmetic and logical expressions. In the last part the implementation details were covered that was used for the prototype implementation. Studying this topic showed that it is possible to use dynamic typing in a statically typed environment and it can be rewarding to use alternative methods to solve computation-intensive problems in the appropriate domains: accessing and processing graph data can be carried out more effectively with connecting Erlang with C++.

REFERENCES

[1] Clean Programming Language. http://wiki.clean.cs.ru.nl/Clean.
[2] Common Object Request Broker Architecture. http://www.corba.org/.
[3] Erlang and C Interoperability. http://www.erlang.org/doc/tutorial/introduction.html.
[4] Erlang Digraph Library. http://www.erlang.org/doc/man/digraph.html.
[5] LEMON Graph Library. http://lemon.cs.elte.hu/trac/lemon.
[6] RefactorErl Homepage. http://plc.inf.elte.hu/erlang.
[7] Standard template library. https://www.sgi.com/tech/stl/.
[8] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically-typed Language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 213–227, New York, NY, USA, 1989. ACM.
[9] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[10] Gordon Blair, Massimo Paolucci, Paul Grace, and Nikolaos Georgantas. Interoperability in Complex Distributed Systems. In Marco Bernardo and Valerie Issarny, editors, *11th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Connectors for Eternal Networked Software Systems*. Springer, 2011.

[11] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl – Source Code Analysis and Refactoring in Erlang. In *In proceeding of the 12th Symposium on Programming Languages and Software Tools, Tallin, Estonia*, 2011.

[12] Tobias Gedell and Daniel Hedin. Abstract Interpretation Plugins for Type Systems. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology*, AMAST 2008, pages 184–198, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Attila Góbi, Zalán Szügyi, and Tamás Kozsik. A C++ pearl – self-referring streams. *Annales Univ. Sci. Budapest., Sect. Comp.*, pages 157–174, 2012.

[14] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Víg, Tamás Nagy, Melinda Tóth, and Roland Király. Modeling semantic knowledge in Erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babes-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, Jul 2009.

[15] Valérie Issarny and Amel Bennaceur. Composing Distributed Systems: Overcoming the Interoperability Challenge. In F. de Boer, M. Bonsangue, E. Giachino, and R. Hähnle, editors, *FMCO 2012*, Lecture Notes in Computer Science, pages 168–196. Springer, 2013.

[16] Nykzn Gaizler J. *Programming Languages*. Kiskapu, 2003.

[17] Tobias Lindahl and Konstantinos Sagonas. Practical Type Inference Based on Success Typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 167–178, New York, NY, USA, 2006. ACM.

[18] Tobias Lindahl and Konstantinos F. Sagonas. TypEr: a type annotator of Erlang code. In *Erlang Workshop*, pages 17–25, 2005.

[19] Simon Marlow and Philip Wadler. A Practical Subtyping System For Erlang. In *In Proceedings of the International Conference on Functional Programming (ICFP '97*, pages 136–149. ACM Press, 1997.

[20] Sven-Olof Nyström. A Soft-typing System for Erlang. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, ERLANG '03, pages 56–71, New York, NY, USA, 2003. ACM.

[21] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[22] Bjarne Stroustrup. *The C++ Programming Language, 4th ed.* Addison-Wesley, 2013.

[23] Laurence Tratt. Dynamically Typed Languages, 2009.

[24] Viktória Zsók Zoltán Horváth, Zoltán Varga. Clean-CORBA Interface for Parallel Functional Programming on Clusters. In *Proceedings of 6th International Conference on Applied Informatics*, pages 27–31, 2004.

Faculty of Informatics, Eötvös Loránd University
E-mail address: {neataai, tothmelinda}@caesar.elte.hu