Hello world

Introduction

Static type checking is generally desirable for programming languages improving software quality, lowering the number of bugs and preventing avoidable errors. The idea is to allow compilers to identify as many issues as possible before actually run the program, and therefore avoid a vast number of trivial bugs, catching them at a very early stage. Despite the fact that, in the general, case interesting properties of programs are undecidable [19], static type checking, within its limits, is an effective and well established technique of program verification. If a compiler can prove that a program is well-typed, then it does not need to perform dynamic safety checks, allowing the resulting compiled binary to run faster.

Jolie [15] is the only language natively supporting microservice architectures and, currently, has dynamic type checking only. A static type system for the language has been exhaustively and formally defined on paper, but still lacks an implementation. The obstacles of programming in a language without a static type analyzer have been witnessed by Jolie developers, especially by newcomers. However, implementing such system is a non trivial task due to technical challenges both of general nature and specific to the language. In this paper, we introduce and describe ongoing work on a static type checker for the Jolie programming language [15]. Our approach follows the formal specification rules as defined in [17]. The project is built as a Java implementation of source code processing and verification via Z3 SMT solver [8] and it has to be intended in the context of our community contribution to the Jolie programming language [5].

Section 1.4 recalls the basic of Jolie and section 1.5 discusses related work. The description of the static type-checking and the system architecture can be found in Section 2, while Section 2.4 draws conclusive remarks and discusses open issues.

0.1   Main points

- The problem
    - Brief Jolie description
    - Static vs dynamic typization
- Why is this important
    - Overall opinions
    - Precise examples

- The solution
  - Overview of the tools and approaches?

Chapter 1

System design

1.1 Theoretical base

This paper builds on top of Julie Meinicke Nielsen's work at the Technical University of Denmark (*"A Type System for the Jolie Language"* [17]) implementing the type checker specification. The thesis represents the theoretical foundation for the type checking of the core fragment of the Jolie language, which excludes recursive types, arrays, subtyping of basic types, faults and deployment instructions such as architectural primitives. The work of Nielsen presents the first attempt at formalizing a static type checker for the core fragment of Jolie, and the typing rules expressed there are the core theory behind our static checker.

1.2 Main points

Here I provide literature review. Unfortunately, I haven't done it yet. I still have to work on the proper list of literature that I actually have used during the development.

   I don't think this part is going to be VERY hard. Moreover, this part should be relatively easy and helpful for increasing the overall volume of the thesis. First of all, I describe the paper with theoretical typization. Maybe I should include here the algorithm description, instead of putting it in the next chapter.

1.3 PSI article continues

1.4 Background

Microservices [9] is an architectural style evolved from Service-Oriented Architectures [11]. According to this approach, applications are composed by small independent building blocks that communicate via message passing. These composing parts are indeed called microservices. This paradigm has seen a dramatic growth in popularity in recent years [16]. Microservices are not limited to a specific technology. Systems can be built using a wide range of technologies and still

fit the approach. In this paper, however, we support the idea that a paradigm-based language would bring benefit to development in terms of simplicity and development cost.

Jolie is the first programming language constructed above the paradigm of microservices: each component is autonomous service that can be deployed separately and operated by running in parallel processes. Jolie comprises formally-specified semantics, inspired by process calculi such as CCS [13] and the π-calculus [14]. As for practical side, Jolie is inspired by standards for Service-Oriented Computing such as WS-BPEL [2]. The composition of both theoretical and practical aspects allows Jolie to be the preferred candidate for the application of modern research methodologies, e.g. runtime adaptation, process-aware web applications, or correctness-by-construction of concurrent software.

The basic abstraction unit of Jolie is the microservice [9]. It is based on a recursive model where every microservice can be easily reused and composed for obtaining, in turn, other microservices. Such approach allows distributed architecture and guarantees simple management of all components, which reduces maintenance and development effort. Microservices communicate and work together by sending messages to each other. In Jolie, messages are represented in tree structure. A variable in Jolie is a path in a data tree and the type of a data tree is a tree itself. Equality of types must therefore be handled with that in mind. Variables are not declared wherefore the manipulation of the program state must be inferred. Communications are type checked at runtime, when messages are sent or received. Type checking of incoming messages is especially relevant, since it could moderate the consequences of errors.

The Jolie language is constructed in three layers: The behavioural layer operates with the internal actions of a process and the communication it performs seen from the processs point of view, the service layer deals with the underlying architectural instructions and the network layer deals with connecting communicating services.

Other workflow languages are capable of expressing orchestration of (micro)services the same way Jolie can do, for example WS-BPEL [2]. WS-BPEL allows developers to describe workflows of services and other communication aspects (such as ports and interfaces), and it has been also shown how dynamic workflow reconfiguration can be expressed [12]. However, WS-BPEL has been designed for high-level orchestration, while programming the internal logic of a single micro-service requires fine-grained procedural constructs. Here it is were Jolie works better.

1.5   Related work

The implementation of a static type checker for Jolie is part of a broader attempt to enhance the language for practical use. Previous work on the type system has been done, however focusing mostly on dynamic type checking. Safina extended the dynamic type system as described in [21], where type choices have been added in order to move computation from a process-driven to a data-driven approach.

The idea to integrate dynamic and static type checking with the introduction of refinement types, verified via SMT solver, has been explored in [22]. The integration of the two approaches allows a scenario where the static verification of

internal services and the dynamic verification of (potentially malicious) external services cooperates in order to reduce testing effort and enhancing security.

The idea of using SMT Solvers for static analysis, in particular in combination with other techniques, has been successfully adopted before for other programming languages, for example LiquidHaskell and F*. LiquidHaskell [10][1] is a notable example of implementation of Liquid Types (*Logically Qualified Data Types*) [20]. It is a static verification technique combining *automated deduction* (SMT solvers), *model checking* (Predicate Abstraction), and *type systems* (Hindley-Milner inference). Liquid Types have been implemented for several other programming languages. The original paper presented an OCaml implementation. F* [1] instead an ML-like functional programming language specifically designed for program verification.The F* type-checker uses a combination of SMT solving and manual proofs to guarantee correctness

Another direction in developing static type checking for Jolie is creating the verified type checker [3] by means of proof assistant instead of SMT solver. Proof assistant is a software tool needed to assist with the development of formal proofs by human-machine collaboration and helps to ascertain the correctness of them. The type checker is expressed as well-typed program with dependent types in Agda [18]. If the types are well formed, all required invariants and properties are described and expressed in the types of the program meaning that the program is correct. This work is currently in progress and evolves in parallel with ours.

---

[1]Online demo at http://goto. ucsd.edu/ rjhala/liquid/haskell/demo/

Chapter 2

Static type-checking implementation

2.1 Overview

The type checker implementation is based on three main system components. One of the components is a parser of Jolie programs, written in Java. It has been created and maintained by the core Jolie team. The latest sources of the parser can be found in the original Jolie GitHub repository[1]. The other two system components of the type checker carry out the core functional part of the type checker. Using an AST[2] that was built by the parser described above, they produce a set of SMT[3] theorems for using Z3 [8] syntax. In the following phase they are being processed by Z3 SMT solver itself. The general idea is to generate a restricting formula for each program statement based on the first appearance of variables and constants, and on their usage. This is achieved by implementing a visiting method for every node type of the AST. The algorithm is described below in details. The overall process is visually represented in the Figure 1.

The approach to models verification using SMT solvers is related to the logic and logical theorems. Logic uses theorems as statements of a formal language. A consistent set of such statements form a model of a Jolie program. As a foundation of this model, a set of basic abstract axioms exists. On top of it, a more case-oriented set of theorems is being constructed. Combined together, they formalize the typing rules for each of Jolie program statements, and for the overall program itself. Thus, the Jolie typing rules are specific cases of logical theorems, or rules, that are constructed in the project. The concept is implied from software verification fundamentals [7].

---

[1] https://github.com/jolie/jolie
[2] Abstract Syntax Tree
[3] Satisfiability Modulo Theories

Here Be A Pretty And Unique Picture

Figure 2.1: Process of Type Checking in the Jolie Verifier

Jolie programs may contain complex expressions with function calls, type choosing statements, implicit type casting, etc. Thus, it is also crucial to consider data structures representing a match between names and expressions, in order to be able to avoid inconsistency and redundancy. The approach to solution of this problems is descibed in the *Algorithm description* section below.

2.1.1 Jolie verifier

The Java program reuses an existing structure of a Visitor pattern that was used in a previous project for formatting Jolie source code [4]. It accepts processed Jolie program source code in the form of AST and performs traversing. For each kind of node the system creates one or more logical formulas written using SMT-LIB [6] syntax, which are then stored into a file on a disk[4]. The verifier targets assignments, conditions, and other cases of variables usage where type consistency can be violated.

2.1.2 SMT Solver

Z3 carries out the functionality of program verification. Z3 is an SMT solver from Microsoft Research [8]. It is positioned as a tool for solving problems that occur in software verification, and software analysis. Given a set of theorems that was previously created by the Java component, Z3 processes it and returns whether this set is satisfiable or not. In case of any contradiction in the set, the solver will signal that the overall model is not satisfiable. This means that the input program is not consistent in terms of types usage.

2.2 Algorithm description

Here I describe all the rules I come up with for creating theorems for node types. I describe the structure of a common Jolie program, how do I solve the problem of nodes isolation during the tree traversing, which rules serve as basis of the whole system and how and why do they work, etc.

I have to create illustrations for the process of transforming some very simple statement in Jolie into z3 theorems. Any big statement probably won't fin into a single illustration so I have to be careful with it.

I also need to visually illustrate the process of keeping track of the context while traversing the tree. It is an important part of the algorithm, and it is usefull to have its visual representation. However, I have no idea yet how to draw it.

2.3 Type-checking theorems

Our objective is to accurately translate Jolie typing rules into SMT statements, therefore allowing static type checking [5]. The foreground activity so far is producing the set of statements for the construct of the behavioural layer of Jolie. The layer describes the internal actions of a process and the communications it

---

[4]At the current implementation state the theorems are collected in a single data element
[5]Please note that, at the moment, not all the rules in [17] have been implemented.

performs seen from the process point of view. The layer is chosen for the first phase of the development because of being the foundation of the syntactical structures of Jolie. Also there is a similarity of the layer with common programming languages in a sense of the abstraction level. So these facts make the behavioural level to be the first entry in the world of Jolie language capabilities.

Here we will illustrate an example of the translation in order to understand the procedure in detail. All statements at the behavioural layer of Jolie are called behaviours. We write $\Gamma \vdash_B B \triangleright \Gamma'$ to indicate a behaviour B, typed with respect to an environment Γ, which updates Γ to Γ' during type checking [17].

The conditional typing statement is the following:

$$\frac{\Gamma \vdash e : bool \quad \Gamma \vdash_B B_1 \triangleright \Gamma' \quad \Gamma \vdash_B B_2 \triangleright \Gamma'}{\Gamma \vdash_B if(e) \ B_1 \ else \ B_2 \triangleright \Gamma'}$$

The typing rule of the *if* statement does not contradict intuition. the statement is typable when its condition expression is boolean, and the execution of both its branches brings the same updates to the environment. This means that the set of matches between expressions and variables with their types remains the same with no difference from a branch choice. This is necessary since it is not possible to predict what branch will be executed at runtime. [6]

The full implementation is available on github [7]. Here we show the Java fragment which builds the corresponding SMT statement.

```java
public void visit(IfStatement n) {
    for (Pair<OLSyntaxNode, OLSyntaxNode> statement :
            n.children()) {
        OLSyntaxNode condition = statement.key();
        OLSyntaxNode body = statement.value();
        check(condition);
        TermReference conditionTerm = usedTerms.pop()
            ;
        writer.writeLine("(assert (hasType " +
            conditionTerm.id + " bool))");
        if (body != null) {
            body.accept(this);
        }
    }
    if (n.elseProcess() != null) {
        n.elseProcess().accept(this);
    }
}
```

The code structure represents basic steps to achieve a record with corresponding SMT statements of the block as a result. Firstly, a condition of the *if* statement is separated from the body. Then the condition is sent to be checked using the same visitor class. Eventually after the last 'recursion' step the condition is put in the stack of terms, which contains any terms (expressions, variables etc.) processed during the checking. So the term corresponding to the condition is expected to be on top of the stack. Then an assertion that

---

[6]The *else* part may also be omitted and B_2 may be replaced by an empty behavior.
[7]https://github.com/innopolis-jolie-smt-typechecker/jolie

says the condition term is boolean is written. Afterwards the body is processed using one of the other overloads of the visitor. These steps can be repeated in case of existence of nested conditional statements. In the end of the method the *else* branch body of the very first *if* is processed if it is present. There is also an important note is that the conditional statement does not impose any other direct type restrictions besides the condition term that is confirmed by the mentioned typing rule. Other implemented nodes can be seen in the source mentioned above.

The Jolie verifier takes some input for processing. Let us consider the following simple piece of Jolie code with a conditional statement:

```
1  a = 2;
2  b = 3;
3  if ( a > b ) {
4    println@Console( a + b )()
5  }else{
6    println@Console( "Hello, world!" )()
7  }
```

In the case everything works, none of the typing rules is violated. Z3 agrees with the opinion and results in 'sat', that means the program state is satisfiable. There is the SMT statements representing the condition processing:

```
1 (declare−const $$__term_id_10 Term)
2 (assert (hasType $$__term_id_10 bool))
3
4 (assert (hasType $$__term_id_10 bool))
```

The first assertion is made based on an expression type determination: the expression $a > b$ is boolean. The second one is imposed by the typing rule: the condition expression must be boolean. In the case there is no contradiction between these two assertions.

If the condition would be replaced with some other type expression the typing rule may be violated. There is the corresponding example case with a replacement of $a > b$:

```
1  a = 2;
2  b = 3;
3  if ( 5 ) {
4    println@Console( a + b )()
5  }else{
6    println@Console( "Hello, world!" )()
7  }
```

And the constructed SMT statements for the condition expression are following:

```
1 (declare−const $$__term_id_10 Term)
2 (assert (hasType $$__term_id_10 int))
3
4 (assert (hasType $$__term_id_10 bool))
```

Now the contradiction between the assertions is notable. The parser decided the expression to be an integer, which is correct. But the restriction on a condition

type from the typing rule simply contradict with the actual type. Consequently Z3 results in 'unsat'. This means that the program state representing the assertion unsatisfiable and incorrect in terms of the considered static type checking analysis.

2.4 Correctness verification

The main tool here is testing. I describe different test cases. Probably, I come up with some automatization of the testing process. Maybe unit tests, or something. This has not been decided yet. I provide listings of programs with corresponding output. I need both small and simple programs (not hello world though), and a couple of complex ones. Code goes as Applications at the end of the thesis.

Conclusions

2.5 Main points

Depending on the final state of the checker at the moment of submitting the paper.

- Either we have fully operational system to check any Jolie program

- Or we have significant progress on the way to full operational system to check nay Jolie program

- say what you've just said (repeat the introduction):
    - why is it important?
    - why this system makes life easier?

- What are the applications?

- What are the future plans?

2.6 PSI article continues

Static type checking allows compilers to identify programming mistakes (for what concerns types) at compile time, i.e. before actually running the program. Therefore a vast number of trivial bugs can be avoided being caught and fixed at a very early stage of the software life-cycle. In this paper, we tackled the problem of static type checking for the Jolie programming language, which natively supports microservice. A static type system for the language has been exhaustively and formally defined on paper, but so far still lacked a full implementation. We introduced our ongoing work on a static type checker and presented some details of the implementation. The type checker prototype, at the moment, consists in a set of rules for the type system expressed in Z3. The actual implementation covers operations such as assignments,logical statements, conditions, primitive terms usage and comparison.

The type checker is already able to validate programs, as it has been shown in this paper. However, it works with certain assumptions. The main assumption is that programs do not contain implicit type casts. The Jolie language allows implicit type casts, however, their behavior is very complex. Handling such situations is an open issue left for future development and future versions. Two other major issues have not been addressed.

Variable types can be changed at runtime.   This strictly depends on the approach that has been chosen. Generally, static typing guarantees that a variable has a type that cannot be changed after declaration or assignment. However, Jolie allows this operation. We need to determine which behavior we expect from the type checker, thus deciding how to process type changes.

Implicit type casts in Jolie are ambiguous.   This is a major problem, and further research is required in order to find a solution. While Jolie allows implicit type casts, sometimes the result of a cast is not obvious. For example, casting a negative Integer to Boolean will result in a False. This is an unexpected behavior when compared to other programming languages. There may be a solid rationale for this, however, we need to investigate all cases and make sure that the type checker works accordingly to the Jolie actual behavior, and not to the expected one.

References

Bibliography

[1] F*. https://www.fstar-lang.org/.

[2] WS-BPEL OASIS Web Services Business Process Execution Language. accessed April 2016. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.

[3] Eugenii Akentev. Verified type checker for Jolie programming language. https://github.com/ak3n/jolie.

[4] Nikita Alekseev. Jolie Code Formatter. https://github.com/nickaleks/jolie.

[5] Alexey Bandura, Nikita Kurilenko, Manuel Mazzara, Victor Rivera, Larisa Safina, and Alexander Tchitchigin. Jolie community on the rise. In *9th IEEE International Conference on Service-Oriented Computing and Applications, SOCA*, 2016.

[6] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard. Version 2.0 , 2010.

[7] Hoare C.A.R. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[8] Leonardo de Moura and Nikolaj Bjrner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[9] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In Bertrand Meyer and Manuel Mazzara, editors, *Present and Ulterior Software Engineering*. Springer, 2017.

[10] Ranjit Jhala. Liquid Haskell. http://goto.ucsd.edu/~rjhala/liquid/haskell/blog/about/.

[11] M.C. MacKenzie et al. Reference model for service oriented architecture 1.0. *OASIS Standard*, 12, 2006.

[12] M. Mazzara, F. Abouzaid, N. Dragoni, and A. Bhattacharyya. Design, modelling and analysis of a workflow reconfiguration. In *Proceedings of the International Workshop on Petri Nets and Software Engineering, Newcastle upon Tyne, UK, June 20-21, 2011*, pages 10–24, 2011.

[13] Robin Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., 1995.

[14] Robin Milner. *Communicating and Mobile Systems: The &Pgr;-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[15] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-Oriented Programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.

[16] S. Newman. *Building microservices*. O'Reilly Media, Inc., 2015.

[17] Julie Meinicke Nielsen. A Type System for the Jolie Language. Master's thesis, Technical University of Denmark, 2013.

[18] Chalmers University of Technology. Accessed December 2016. Agda. `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[19] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[20] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, June 2008.

[21] Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices (genericity in jolie). In *Proc. of The 30th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, 2016.

[22] Alexander Tchitchigin, Larisa Safina, Manuel Mazzara, Mohamed Elwakil, Fabrizio Montesi, and Victor Rivera. Refinement types in jolie. In *Spring/-Summer Young Researchers Colloquium on Software Engineering, SYR-CoSE*, 2016.

Applications