Verified type checker for Jolie programming language

Evgenii Akentev[1], Alexander Tchitchigin[2], Larisa Safina[1,3], Manuel Mazzara[1]

[1] Innopolis University, Russian Federation
ak3ntev@gmail.com, {l.safina, m.mazzara}@innopolis.ru
[2] Innosoft LLC
sad.ronin@gmail.com
[3] University of Southern Denmark

Abstract. Jolie is a service-oriented programming language which comes with the formal specification of its type system. However, there is no tool to ensure that programs in Jolie are well-typed. In this paper we provide the results of building a type checker for Jolie as a part of its syntax and semantics formal model. We express the type checker as a program with dependent types in Agda proof assistant which helps to ascertain that the type checker is correct.

Keywords: formal verification, type checker, dependent types, Agda, Jolie, type systems, microservices

1 Introduction

Microservices architecture is a modern paradigm in software development using a composition of autonomous entities for creating systems [1]. It has been developed as the answer to the problems arisen in applications built in monolith or Service-Oriented Architecture styles including difficulties with scalability, complexity and dependencies of the evolving application. Microservices implement only a limited and cohesive amount of functionality, run their own processes, and use lightweight communication mechanisms.

In the fast growing landscape of microservices, Jolie [2] appears to be a good candidate to play the role of paradigmatic programming language [3]. Since every program in Jolie is a microservice, everything can be reused or recomposed for obtaining new microservices making easy creation of as simple services as complex architectural compositions. This makes programs to scale easily, thus supports distributed architecture with simple managing of components, reducing maintenance and lower development costs.

However, communication between microservices in Jolie is obtained by means of sending and receiving messages whose types correspondence is checked only at runtime. Having a formalized type system of programming language gives us an opportunity to implement type checking mechanism and use it before the runtime.

Our idea is to augment Jolie with static type-checking mechanism based on type system specification. Type system of Jolie, described by Nielsen in [4], represents typing rules for the core of programming language (excluding subtyping, recursive types and some other primitives) and gives as a theoretical basis to reason about correctness of a program in Jolie.

We decided to use Agda [5,6] as a proof assistant for implementing our type checker. Agda is a functional programming based on dependent types. Agda represents ans extension to the Martin-Löf's logical framework [7,8]. Thus we can introduce logical propositions as types by means of Curry-Howard isomorphism [9] and prove them writing the type corresponding programs. Agda possesses concrete syntax and comes with rich family of data types, pattern matching mechanism, termination checking, as well as with the ordinary programming constructs.

The paper has the following structure. Section 2 provides a background for Jolie programming language and presents a the subset of its formalization[4] written in Agda. We implement the syntax of the behavioral layer of Jolie, which describes the workflow of service activities, in section 2, and in section 3 we provide the necessary subset of typing rules. Section 4 contains the proof of "Structural Congruence" lemma for behaviours showing the correctness of rules introduced. Finally, in section 5 we conclude our paper and describe the possible directions of future work.

2 Jolie Formalization

Formal syntax and semantic of Jolie are based on SOCK process calculi [11,12]. SOCK was created for designing service-oriented systems and was inspired by notable π-calculus [13] and WS-BPEL [14]. Primitives in SOCK are able to express one-way and request-response communication, parallel and sequential behavior of processes and control primitives.

SOCK (so a formalization of Jolie program) comprises of three layers:

- *Behavioral layer*: specifies with internal actions of a process and communication performs as seen from the process' point of view.
- *Service layer*: it deals with underlying architectural instructions, states, service instances and correlation sets.
- *Network layer*: is in charge of connecting and interacting of communicating services.

At the current stage of research we have formalized the behavior level of Jolie. We present our results and the detailed description of the behavioral level in the following subsection.

2.1 Syntax of the behavioral layer

The most important type of statements in behavioral level regards performing communications and handling data.

Communications There are two types of communication statements in Jolie: input and output statements, both can be uni-directional (one-way operations) and bi-directional (request-response operations). In case of output statements, we use the notion of an output port name (location) which is necessary for binding the communicated data to it. The data is made use by communication statements as variable paths and expressions described below.

[4] The whole formalization is available here [10]

Handling data Communication messages in Jolie are represented by means of variable paths structured as a tree. For example:

$$\begin{gathered}\text{amount} = 12\\ \text{amount.fruit.apple} = 2\\ \text{amount.fruit.description} = \text{"Apple"}\end{gathered}$$

To simplify further operations with variables, we propose their enumeration. Let J be a Jolie program, $V = vars(J)$ – variables in J, then $V_i = i$ where $i \in \mathbb{N}$. Then the example above will look like:

$$\begin{gathered}0 = 12\\ 1 = 2\\ 2 = \text{"Apple"}\end{gathered}$$

After this simplification the type of variables can be defined. The type of natural numbers is located in standard library of Agda [15].

```
Variable : Set
Variable = ℕ
```

Complete syntax of behavioral layer can be found in [4]. We do not need to consider expressions' structure to prove desired theorems therefore type `Expr` is left empty.

```
data Expr : Set where
```

Operation names, channel names and locations are represented by strings.

```
Operation Location Channel : Set
Operation = String
Location = String
Channel = String
```

The behavioural layer has both ordinary control–flow statements ('if-then-else', 'while', 'assign') and special statements to control parallelism and communication ('inputchoice', 'parallel', 'input', 'output', etc).

```
data Behaviour : Set where
  if_then_else_ : Expr → Behaviour → Behaviour → Behaviour
  while[_]_ : Expr → Behaviour → Behaviour

  -- Sequence
  _⇒_ : Behaviour → Behaviour → Behaviour

  -- Parallel
  _‖_ : Behaviour → Behaviour → Behaviour

  -- Assign
```

```
  _≃_ : Variable → Expr → Behaviour

  nil : Behaviour

  -- [η₁]{B₁}⋯[ηₐ]{Bₐ}
  inputchoice : List (η × Behaviour) → Behaviour

  wait : Channel → Operation → Location → Variable → Behaviour
  exec : Channel → Operation → Variable → Behaviour → Behaviour

  input : η → Behaviour
  output : η^ → Behaviour

-- Input
data η where
  -- o(x) -- One-way
  _[_] : Operation → Variable → η

  -- o(x)(x'){B} -- Request-response
  _[_][_]_ : Operation → Variable → Variable → Behaviour → η

-- Output
data η^ where
  -- o@l(e) -- Notification
  _at_[_] : Operation → Location → Expr → η^

  -- o@l(e)(x) -- Solicit-response
  _at_[_][_] : Operation → Location → Expr → Variable → η^
```

3 Jolie type system

Jolie type system consists of commonly-used native types as `int`, `double`, `long`, `boolean` and `string`, Jolie also has the following types:

- `raw`, representing raw data streams as byte arrays.
- `void`, indicating no value.
- `any`, as a placeholder for any native types.

In order to be able to do type checking of a Jolie program, we need to provide implementation of types and typing rules in Agda.

3.1 Type declaration

We express main Jolie native types (excluding any) in the following way:

```
data Type : Set where
  bool int double long string raw void : Type
```

Usually, the context of a program is a list of variables, but to service all three layers (comprising communication of services) there is a special type called `TypeDecl`. It has five constructors: the first two (unidirectional and bidirectional) are for output communication. The left part of such bindings consists of an operation name and a location of a hosting service. The next two are for input communication and the last one is for variables.

```
data TypeDecl : Set where
  - o@l : <T>
  _at_:<_> : Operation → Location → Type → TypeDecl

  - o@l : <T, T>
  _at_:<_,_> : Operation → Location → Type → Type → TypeDecl

  - o : <T>
  _:<_> : Operation → Type → TypeDecl

  - o : <T, T>
  _:<_,_> : Operation → Type → Type → TypeDecl

  - x : T
  _:_ : Variable → Type → TypeDecl
```

Therefore, the type of context is a vector of `TypeDecl`.

```
Ctx : ℕ → Set
Ctx = Vec TypeDecl
```

Although the type of context is defined, it's not enough, because programs in Jolie can be parallel. We define one more type called `Context` to cover such situations. It has only two constructors: the first one just takes `Ctx n` and the second one consists of two elements of itself.

```
data Context : Set where
  ★ : ∀ {n} → Ctx n → Context
  & : Context → Context → Context
```

The type of context is not a vector anymore, so we need to define such type that will express the fact of presence of `TypeDecl` in `Context`.

```
infix 4 _∈_
data _∈_ : TypeDecl → Context → Set where
  here–★ : ∀ {n} {x} {xs : Ctx n}
    → x ∈ ★ (x :: xs)
```

```
there-★ : ∀ {n} {x y} {xs : Ctx n}
  (x∈xs : x ∈ ★ xs)
→ x ∈ ★ (y :: xs)

here-left-& : ∀ {n m} {x} {xs : Ctx n} {ys : Ctx m}
  → x ∈ &  (★ (x :: xs)) (★ ys)

here-right-& : ∀ {n m} {x} {xs : Ctx n} {ys : Ctx m}
  → x ∈ & (★ xs) (★ (x :: ys))

there-left-& : ∀ {n m} {x} {xs : Ctx n} {ys : Ctx m}
  (x∈xs : x ∈ & (★ xs) (★ ys))
  → x ∈ & (★ (x :: xs)) (★ ys)

there-right-& : ∀ {n m} {x} {xs : Ctx n} {ys : Ctx m}
  (x∈xs : x ∈ & (★ xs) (★ ys))
  → x ∈ & (★ xs) (★ (x :: ys))
```

Since we don't care about expressions at all, we introduce the empty type of a correctly typed expression with variables from context Γ.

```
data _⊢e_:_ (Γ : Context) : Expr → Type → Set where
```

3.2 Typing rules

Finally, we can present the subset of the typing rules of the behavioural layer. The first constructor is for `nil` behaviour. Since `nil` does nothing, the contexts before and after are equal. The next two are rules for ordinary behaviours `if_then_else` and `while`. Finally, the last two are for sequent and parallel statements.

```
data _⊢B_▷_ : Context → Behaviour → Context → Set where
  t-nil : {Γ : Context}
    → Γ ⊢B nil ▷ Γ

  t-if : {Γ Γ₁ : Context} {b₁ b₂ : Behaviour} {e : Expr}
    → Γ ⊢e e : bool
    → Γ ⊢B b₁ ▷ Γ₁
    → Γ ⊢B b₂ ▷ Γ₁
    → Γ ⊢B if e then b₁ else b₂ ▷ Γ₁

  t-while : {Γ : Context} {b : Behaviour} {e : Expr}
    → Γ ⊢e e : bool
    → Γ ⊢B b ▷ Γ
    → Γ ⊢B while[ e ] b ▷ Γ
```

```
t−seq : {Γ Γ₁ Γ₂ : Context} {b₁ b₂ : Behaviour}
  → Γ ⊢B b₁ ▷ Γ₁
  → Γ₁ ⊢B b₂ ▷ Γ₂
  → Γ ⊢B b₁ ⇒ b₂ ▷ Γ₂

t−par : {Γ₁ Γ₂ Γ₁′ Γ₂′ : Context} {b1 b2 : Behaviour}
  → Γ₁ ⊢B b1 ▷ Γ₁′
  → Γ₂ ⊢B b2 ▷ Γ₂′
  → (& Γ₁ Γ₂) ⊢B b1 ∥ b2 ▷ (& Γ₁′ Γ₂′)
```

4 Structural Congruence for Behaviours

According to the Curry-Howard isomorphism [9], types of the programs are propostions and terms are proofs. For example, the type $A \to B$ correspond to the implication from A to B and such function f that takes an element of type A and returns an element of type B will be a proof of this theorem.

To demonstrate the correctness of the typing rules given above, we will prove the lemma called "Structural Congruence for Behaviours" [4,16]:

$$\begin{gathered} \mathit{Let}\ \Gamma \vdash B_1 \rhd \Gamma' \\ \mathit{If}\ B_1 \equiv B_2 \\ \mathit{then}\ \Gamma \vdash B_2 \rhd \Gamma' \end{gathered}$$

The proof is the case analysis of all possible B_1 and B_2.

– *Case* $B_1 \equiv B_2$

```
struct−cong−b₁≡b₂ : {Γ Γ₁ : Context} {b₁ b₂ : Behaviour}
  → Γ ⊢B b₁ ▷ Γ₁
  → b₁ ≡ b₂
  → Γ ⊢B b₂ ▷ Γ₁
struct−cong−b₁≡b₂ t refl = t
```

– *Case* $0; B \equiv B$

```
struct−cong−nil:b→b : {Γ Γ₁ : Context} {b : Behaviour}
  → Γ ⊢B nil ⇒ b ▷ Γ₁
  → Γ ⊢B b ▷ Γ₁
struct−cong−nil:b→b (t−seq t−nil x) = x
```

– *Case* $B \equiv 0; B$

```
struct−cong−b→nil:b : {Γ Γ₁ : Context} {b : Behaviour}
  → Γ ⊢B b ▷ Γ₁
  → Γ ⊢B nil ⇒ b ▷ Γ₁
struct−cong−b→nil:b x = t−seq t−nil x
```

– *Case* $B \parallel 0 \equiv B$

```
struct−cong−b‖nil→b : {Γ1 Γ2 Γ1′ Γ2′ : Context} {b : Behaviour}
  → & Γ1 Γ2 ⊢B (b ‖ nil) ▷ & Γ1′ Γ2′
  → Γ1 ⊢B b ▷ Γ1′
struct−cong−b‖nil→b (t−par x _) = x
```

– *Case* $B \equiv B \parallel 0$

```
struct−cong−b→b‖nil : {Γ1 Γ2 Γ3 : Context} {b : Behaviour}
  → Γ1 ⊢B b ▷ Γ2
  → & Γ1 Γ3 ⊢B (b ‖ nil) ▷ & Γ2 Γ3
struct−cong−b→b‖nil x = t−par x t−nil
```

– *Case* $B_1 \parallel B_2 \equiv B_2 \parallel B_1$

```
struct−cong−par−comm : {Γ1 Γ2 Γ1′ Γ2′ : Context} {b1 b2 : Behaviour}
  → & Γ1 Γ2 ⊢B (b1 ‖ b2) ▷ & Γ1′ Γ2′
  → & Γ2 Γ1 ⊢B (b2 ‖ b1) ▷ & Γ2′ Γ1′
struct−cong−par−comm (t−par t1 t2) = t−par t2 t1
```

– *Case* $(B_1 \parallel B_2) \parallel B_3 \equiv B_1 \parallel (B_2 \parallel B_3)$

```
struct−cong−par−assoc : {Γ1 Γ2 Γ3 Γ1′ Γ2′ Γ3′ : Context} {b1 b2 b3 : Behaviour}
  → & (& Γ1 Γ2) Γ3 ⊢B (b1 ‖ b2) ‖ b3 ▷ & (& Γ1′ Γ2′) Γ3′
  → & Γ1 (& Γ2 Γ3) ⊢B b1 ‖ (b2 ‖ b3) ▷ & Γ1′ (& Γ2′ Γ3′)
struct−cong−par−assoc (t−par (t−par t1 t2) t3) = t−par t1 (t−par t2 t3)
```

The proof for $B_1 \parallel (B_2 \parallel B_3) \equiv (B_1 \parallel B_2) \parallel B_3$ is similar.

5 Conclusions and future work

In this paper, we presented our approach in creating the static type-checker for Jolie programming language which is currently dynamically type-checked. We developed the formalization for the subset of Jolie by means of Agda proof assistant. We expressed Jolie types and typing rules in order to be able to check the type correspondence of messages which are used for interaction of microservices in Jolie. We have also provided the proof of structural congruence lemma which means the correctness of type checker itself.

However, our current implementation covers only the small subset of Jolie. We have touched only the native types, though the type system of Jolie goes beyond it and includes subtyping, linked types etc. Another important direction leads us to formalization of the service and communication levels of Jolie. By accomplishing this, we would be able to type-check Jolie program thoroughly and may even think about implementing of verifiable compiler for Jolie similar to [17,18] .

References

1. Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *CoRR*, abs/1606.04036, 2016.
2. Fabrizio Montesi. JOLIE: a Service-oriented Programming Language. Master's thesis, University of Bologna, 2010.
3. Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-Oriented Programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.
4. Julie Meinicke Nielsen. A type system for the jolie language. *Technical University of Denmark Informatics and Mathematical Modelling*, 2013.
5. Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda — a functional language with dependent types. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 73–78, Berlin, Heidelberg, 2009. Springer-Verlag.
6. Chalmers University of Technology. Accessed December 2016. Agda. `http://wiki.portal.chalmers.se/agda/pmwiki.php`.
7. Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Studies in proof theory. Bibliopolis, Napoli, 1984.
8. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, USA, 0 edition, July 1990.
9. Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., New York, NY, USA, 2006.
10. Eugenii Akentev. Verified type checker for Jolie programming language. `https://github.com/ak3n/jolie`.
11. Claudio Guidi. *Formalizing languages for Service Oriented Computing*. PhD thesis, 2007.
12. Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. Sock: A calculus for service oriented computing. In *Proceedings of the 4th International Conference on Service-Oriented Computing*, ICSOC'06, pages 327–338, Berlin, Heidelberg, 2006. Springer-Verlag.
13. Robin Milner. *Communicating and Mobile Systems: The &Pgr;-calculus*. Cambridge University Press, New York, NY, USA, 1999.
14. WS-BPEL OASIS Web Services Business Process Execution Language. accessed April 2016. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html`.
15. The agda standard library. `https://github.com/agda/agda-stdlib`.
16. J. Engelfriet, T. Gelsema, and Rijksuniversiteit te Leiden. LIACS Leiden Institute of Advanced Computer Science. *Structural Congruence in the Pi-calculus with Potential Replication*. Technical report LIACS Leiden Institute of Advanced Computer Science. Leiden University, 2000.
17. Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. *SIGPLAN Not.*, 50(9):166–178, August 2015.
18. Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.