Verified type checker for Jolie programming language

Evgenii Akentev\textsuperscript{1}, Alexander Tchitchigin\textsuperscript{2}, Larisa Safina\textsuperscript{1,3}, Manuel Mazzara\textsuperscript{1}

\textsuperscript{1} Innopolis University, Russian Federation
ak3ntev@gmail.com, \{l.safina, m.mazzara\}@innopolis.ru
\textsuperscript{2} Innosoft LLC
sad.ronin@gmail.com
\textsuperscript{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 \cite{5,6} as a proof assistant for implementing our type checker. Agda is a functional programming based on dependent types. Agda represents an extension to the Martin-Löf’s logical framework \cite{7,8}. Thus we can introduce logical propositions as types by means of Curry-Howard isomorphism \cite{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 \ref{section2} provides a background for Jolie programming language and presents a subset of its formalization written in Agda. We implement the syntax of the behavioral layer of Jolie, which describes the workflow of service activities, in section \ref{section2} and in section \ref{section3} we provide the necessary subset of typing rules. Section \ref{section4} contains the proof of "Structural Congruence" lemma for behaviours showing the correctness of rules introduced. Finally, in section \ref{section5} we conclude our paper and describe the possible directions of future work.

\section{Jolie Formalization}

Formal syntax and semantic of Jolie are based on SOCK process calculi \cite{11,12}. SOCK was created for designing service-oriented systems and was inspired by notable \(\pi\)-calculus \cite{13} and WS-BPEL \cite{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:

- \textit{Behavioral layer}: specifies with internal actions of a process and communication performs as seen from the process’ point of view.
- \textit{Service layer}: it deals with underlying architectural instructions, states, service instances and correlation sets.
- \textit{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.

\subsection{Syntax of the behavioral layer}

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

\textbf{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.

\footnote{The whole formalization is available here \cite{10}}
**Handling data** Communication messages in Jolie are represented by means of variable paths structured as a tree. For example:

```
amount = 12
amount.fruit.apple = 2
amount.fruit.description = "Apple"
```

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:

```
0 = 12
1 = 2
2 = "Apple"
```

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

```latex
\text{Variable} : \text{Set} \\
\text{Variable} = \mathbb{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 $\text{Expr}$ is left empty.

```latex
\text{data} \text{Expr} : \text{Set where}
```

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

```latex
\text{Operation} \text{Location} \text{Channel} : \text{Set} \\
\text{Operation} = \text{String} \\
\text{Location} = \text{String} \\
\text{Channel} = \text{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).

```latex
\text{data} \text{Behaviour} : \text{Set where}
```

- **Sequence**
  - $\Rightarrow : \text{Behaviour} \rightarrow \text{Behaviour} \rightarrow \text{Behaviour}$

- **Parallel**
  - $\parallel : \text{Behaviour} \rightarrow \text{Behaviour} \rightarrow \text{Behaviour}$

- **Assign**
\_\sim\_ : \text{Variable} \rightarrow \text{Expr} \rightarrow \text{Behaviour}

\text{nil} : \text{Behaviour}

- \{\eta_1\} B_1 \cdots \{\eta_a\} B_a

\text{inputchoice} : \text{List} (\eta \times \text{Behaviour}) \rightarrow \text{Behaviour}

\text{wait} : \text{Channel} \rightarrow \text{Operation} \rightarrow \text{Location} \rightarrow \text{Variable} \rightarrow \text{Behaviour}

\text{exec} : \text{Channel} \rightarrow \text{Operation} \rightarrow \text{Variable} \rightarrow \text{Behaviour} \rightarrow \text{Behaviour}

\begin{align*}
\text{input} : \eta &\rightarrow \text{Behaviour} \\
\text{output} : \eta^\wedge &\rightarrow \text{Behaviour}
\end{align*}

- \text{Input}

\begin{align*}
\text{data} \eta \text{ where} \\
&\quad \text{o}(x) \rightarrow \text{One-way} \\
&\quad \_\_\_ : \text{Operation} \rightarrow \text{Variable} \rightarrow \eta
\end{align*}

- \text{o}(x)(x')\{B\} \rightarrow \text{Request-response}

\begin{align*}
&\quad \_\_\_\_\_\_ : \text{Operation} \rightarrow \text{Variable} \rightarrow \text{Variable} \rightarrow \text{Behaviour} \rightarrow \eta
\end{align*}

- \text{Output}

\begin{align*}
\text{data} \eta^\wedge \text{ where} \\
&\quad \text{o}@l(e) \rightarrow \text{Notification} \\
&\quad \_\_\_\_\_\_ \_\_ : \text{Operation} \rightarrow \text{Location} \rightarrow \text{Expr} \rightarrow \eta^\wedge
\end{align*}

- \text{o}@l(e)(x) \rightarrow \text{Solicit-response}

\begin{align*}
&\quad \_\_\_\_\_\_\_\_ : \text{Operation} \rightarrow \text{Location} \rightarrow \text{Expr} \rightarrow \text{Variable} \rightarrow \eta^\wedge
\end{align*}

3 Jolie type system

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

- \text{raw}, representing raw data streams as byte arrays.
- \text{void}, indicating no value.
- \text{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 ∈ & (⋆ xs) (⋆ (x∷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_:_身 (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_:_身 : Context → Behaviour → Context → Set where

t→nil : [Γ : Context]
→ Γ ⊢ B nil ⊝ Γ

t→if : [Γ Γ_1 : Context] [b_1 b_2 : Behaviour] {e : Expr}
→ Γ ⊢ e : bool
→ Γ ⊢ B b_2 ⊝ Γ_1
→ Γ ⊢ B b_2 ⊝ Γ_1
→ Γ ⊢ B if e then b_1 else b_2 ⊝ Γ_1

t→while : [Γ : Context] [b : Behaviour] {e : Expr}
→ Γ ⊢ e : bool
→ Γ ⊢ B b ⊝ Γ
→ Γ ⊢ B while [e] b ⊝ Γ
Structural Congruence for Behaviours

According to the Curry-Howard isomorphism \([9]\), types of the programs are propositions and terms are proofs. For example, the type \(A \rightarrow B\) 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" \([4,16]\):

\[
\text{Let } \Gamma \vdash B_1 \triangleright \Gamma' \\
\text{If } B_1 \equiv B_2 \\
\text{then } \Gamma \vdash B_2 \triangleright \Gamma'
\]

The proof is the case analysis of all possible \(B_i\) and \(B_j\).

- Case \(B_1 \equiv B_2\)

\[
\text{struct-cong-b}\equiv b_2 : \{\Gamma \Gamma_j : \text{Context} \} \{b_j b_2 : \text{Behaviour}\} \\
\rightarrow \Gamma \vdash B b_1 \triangleright \Gamma_j \\
b_1 \equiv b_2 \\
\rightarrow \Gamma \vdash B b_2 \triangleright \Gamma_j \\
\text{struct-cong-b}\equiv b_2 \triangleright t \text{ refl } = t
\]

- Case \(0; B \equiv B\)

\[
\text{struct-cong-nil:b\rightarrow b} : \{\Gamma \Gamma_i : \text{Context} \} \{b : \text{Behaviour}\} \\
\rightarrow \Gamma \vdash B \text{nil} \triangleright b \triangleright \Gamma_i \\
\rightarrow \Gamma \vdash B b \triangleright \Gamma_i \\
\text{struct-cong-nil:b\rightarrow b} (t\text{-seq t\text{-nil}} \triangleright x) = x
\]

- Case \(B \equiv 0; B\)

\[
\text{struct-cong-b\rightarrow nil:b} : \{\Gamma \Gamma_i : \text{Context} \} \{b : \text{Behaviour}\} \\
\rightarrow \Gamma \vdash B b \triangleright \Gamma_i \\
\rightarrow \Gamma \vdash B \text{nil} \triangleright b \triangleright \Gamma_i \\
\text{struct-cong-b\rightarrow nil:b} : x = t\text{-seq t\text{-nil}} \triangleright x
\]
- Case $B \parallel 0 \equiv B$

\[
\begin{align*}
\text{struct} &- \text{cong} - \text{b} \parallel \text{nil} \rightarrow \text{b} : \Gamma_1 \Gamma_2 : \text{Context} \Rightarrow [b : \text{Behaviour}] \\
& \quad \rightarrow \Gamma_1 \Gamma_2 \vdash B (b \parallel \text{nil}) \triangleright \Gamma_1' \Gamma_2' \\
& \quad \rightarrow \Gamma_1 \vdash b \triangleright \Gamma_1' \\
\text{struct} &- \text{cong} - \text{b} (t \mathbin{\parallel \text{nil}}) = x
\end{align*}
\]

- Case $B \parallel 0$

\[
\begin{align*}
\text{struct} &- \text{cong} - \text{b} \parallel \text{nil} : \{ \Gamma_1 \Gamma_2 \Gamma_3 : \text{Context} \} \Rightarrow [b : \text{Behaviour}] \\
& \quad \rightarrow \Gamma_1 \Gamma_2 \vdash B (b \parallel \text{nil}) \triangleright \Gamma_2 \\
& \quad \rightarrow \Gamma_1 \Gamma_3 \vdash B (b \parallel \text{nil}) \triangleright \Gamma_1' \Gamma_3 \\
\text{struct} &- \text{cong} - \text{b} (t \mathbin{\parallel \text{nil}}) x = t \mathbin{\parallel \text{nil}}
\end{align*}
\]

- Case $B_1 \parallel B_2 \equiv B_1 \parallel B_1$

\[
\begin{align*}
\text{struct} &- \text{cong} - \text{par} - \text{comm} : \{ \Gamma_1 \Gamma_2 \Gamma_3 \Gamma_1' \Gamma_2' \Gamma_3' : \text{Context} \} \Rightarrow [b_1 b_2 : \text{Behaviour}] \\
& \quad \rightarrow \Gamma_1 \Gamma_2 \Gamma_3 \vdash B (b_1 \parallel b_2 \triangleright \Gamma_1' \Gamma_2' \Gamma_3') \\
& \quad \rightarrow \Gamma_2 \Gamma_3 \vdash B (b_2 \parallel b_1 \triangleright \Gamma_2' \Gamma_1' \Gamma_3') \\
\text{struct} &- \text{cong} - \text{par} - \text{comm} (t \mathbin{\parallel \text{par}} t_1 t_2) = t \mathbin{\parallel \text{par}} t_2 t_1
\end{align*}
\]

- Case $(B_1 \parallel B_2) \parallel B_3 \equiv B_1 \parallel (B_2 \parallel B_3)$

\[
\begin{align*}
\text{struct} &- \text{cong} - \text{par} - \text{assoc} : \{ \Gamma_1 \Gamma_2 \Gamma_3 \Gamma_1' \Gamma_2' \Gamma_3' : \text{Context} \} \Rightarrow [b_1 b_2 b_3 : \text{Behaviour}] \\
& \quad \rightarrow \Gamma_1 \Gamma_2 \Gamma_3 \vdash B (b_1 \parallel b_2 \parallel b_3 \triangleright \Gamma_1' \Gamma_2' \Gamma_3') \\
& \quad \rightarrow \Gamma_1 \Gamma_2 \Gamma_3 \vdash B (b_2 \parallel b_3 \parallel b_1 \triangleright \Gamma_1' \Gamma_2' \Gamma_3') \\
\text{struct} &- \text{cong} - \text{par} - \text{assoc} (t \mathbin{\parallel \text{par}} (t_1 \mathbin{\parallel \text{par}} t_2) t_3) = t \mathbin{\parallel \text{par}} t_1 (t \mathbin{\parallel \text{par}} t_2 t_3)
\end{align*}
\]

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