Certified Compilation of Choreographies with `hacc`

Luís Cruz-Filipe$^{[0000-0002-7866-7484]}$, Lovro Lugović$^{[0000-0001-9684-9567]}$, and Fabrizio Montesi$^{[0000-0003-4666-901X]}$

Department of Mathematics and Computer Science, University of Southern Denmark
`{lcf, lugovic, fmontesi}@imada.sdu.dk`

Abstract. Programming communicating processes is challenging, because it requires writing separate programs that perform compatible send and receive actions at the right time during execution. Leaving this task to the programmer can easily lead to bugs. *Choreographic programming* addresses this challenge by equipping developers with high-level abstractions for codifying the desired communication structures from a global viewpoint. Given a choreography, implementations of the involved processes can be automatically generated by *endpoint projection (EPP)*. While choreographic programming prevents manual mistakes in the implementation of communications, the correctness of a choreographic programming framework crucially hinges on the correctness of its complex compiler, which has motivated formalisation of theories of choreographic programming in theorem provers. In this paper, we build upon one of these formalisations to construct a toolchain that produces executable code from a choreography.

Keywords: Choreographic programming · Certified compilation · Jolie · Formal verification

1 Introduction

In traditional distributed programming, the programmer is tasked with writing the implementation of each *process* (endpoint) as a separate program, taking care of correctly matching send and receive actions in the different programs. This approach is known to be cumbersome and error-prone [7].

In *Choreographic programming* [8], developers specify the desired communications between processes from a global viewpoint. Given a choreographic program (called *choreography*), correct implementations for all involved processes can be automatically generated by a procedure known as *endpoint projection (EPP)* [9]. This avoids manual mistakes in the programming of communication actions, and provides important theoretical advantages, like deadlock-freedom by design—distributed code generated from a choreography is always deadlock-free, as choreographic languages do not have syntax for unmatched communications [1]. In addition to these correctness advantages, this also saves time and lets the programmer focus on the bigger picture of the protocol being developed.

Defining and implementing EPP is technically involved [13], which motivated mechanising theories of choreographic programming using interactive theorem

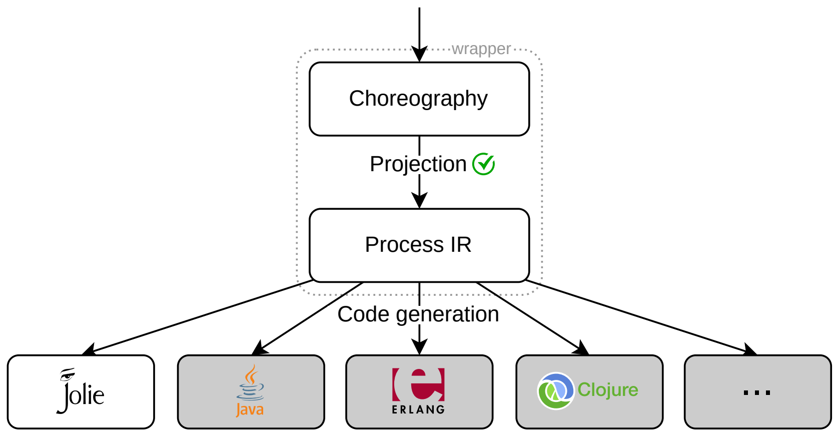

Fig. 1. The architecture of `hacc`'s compilation pipeline. In this work we target the Jolie programming language, but the architecture is designed with extensibility in mind. The languages with a grey background are given as examples of potential future targets.

provers [3,4,6,12]. Among these, the formalisation of *Core Choreographies (CC)* and its EPP to the process calculus of *Stateful Processes (SP)* was designed with broad applicability in mind [5]. Specifically, its design allows for annotating communications with arbitrary metadata and is parametric on the languages used to express local computation, data, process identifiers, etc.

In this work, for the first time, we reap the benefits of [5] to develop `hacc` (Haskell Core Choreographies, pronounced "hack"): a tool for compiling choreographies in CC to executable code. As target language for this executable code, we use the service-oriented programming language Jolie [10]. Additionally, `hacc` is designed with extensibility in mind, so that new target languages can be added.

The architecture of `hacc` consists of two compilation phases (Figure 1). The first phase (projection) uses EPP to translate a choreography in CC into an abstract representation of process programs given in SP, used as an intermediate representation (IR). This phase is certified: it uses a Haskell program extracted from the Coq formalisation [3]. The second phase (code generation) translates the abstract actions of SP into an executable programming language. This phase is not formally verified: since it is a homomorphic transformation that follows the term structure of SP in a straightforward fashion, its correctness is easy to establish directly by manual inspection.

Our development allows for writing and executing CC choreographies [2,3,4,9]. Also, it confirms the informal claim that the formalisation of CC was made with flexibility in mind for this kind of applications [5]. In particular, our application to Jolie did not require any modification of the formalisation, but just an appropriate instantiation of its parameters and a simple interfacing of its extracted data types with the other components of our compilation pipeline in Haskell.

Structure. Section 2 gives an overview of our compiler's architecture and describes its implementation. Section 3 explains how we map the process language to Jolie in order to generate executable Jolie code consisting of multiple independent services.

Related work. There are two other formalisations of choreographic programming. Kalas [12] is a choreographic programming language formalised in the Hol4 proof assistant. It comes with an end-to-end certified compiler that targets CakeML [11], a formally-verified subset of the ML language. By contrast, we take a technology-agnostic approach that allows for reusing our compiler infrastructure for different target languages. Pirouette [6] is a functional choreographic programming language formalised in Coq, which similarly to [4] can be instantiated with different languages for local computation. However, it has not yet been used to implement a choreographic compiler that targets executable code.

2 Choreographies in `hacc`

We represent choreographies and processes (terms of CC and SP, respectively) by Haskell data types that have been automatically extracted from the Coq formalisation. However, a few considerations are necessary, given that Coq is a dependently-typed language, while all of the languages it can extract to are not. As mentioned before, the formalisation is parametric over the types used to represent process identifiers, (recursion) variables, terms of the local computation language, etc. Because this is done using the dependently-typed features of Coq, the extracted Haskell code has some peculiarities.

In particular, instead of using Haskell's type system and parametric polymorphism, the extracted code uses Haskell's `Any` type to achieve genericity. As a consequence, interfacing the extracted code requires using the `unsafeCoerce` function. To deal with the verbosity we provide a more ergonomic interface with a thin *wrapper* (Figure 1) around the extracted code. The wrapper hides the necessary coercions and models the terms using Haskell's parametric polymorphism. This step is not formally verified, but the wrapper again follows the term structure of CC and SP, so checking it for correctness manually is not a problem.

Figure 2 shows our interface. We fix the types of process identifiers (`Pid`), variable names (`Var`, `RecVar`), selection labels (`Label`) and annotations (`Ann`) for simplicity. The `Label` type is a binary sum type with `CLeft` and `CRight` as its constructors, while others are wrappers around **`String`**s, used as identifiers.

Choreographies have type `Choreography e b`, parametric on the local computation and Boolean expression languages (`e` and `b`). Processes can perform point-to-point interactions (`Interaction`)—value communications (`Com`), where a process evaluates an expression and sends the result to another, or selections (`Sel`), where a process selects how another process should behave by communicating a label. Interactions include an annotation (`Ann`), discussed below. Conditionals

```haskell
data Eta e = Com Pid e Pid Var | Sel Pid Pid Label

data Choreography e b
  = CEnd
  | Interaction (Eta e) Ann (Choreography e b)
  | CCond Pid b (Choreography e b) (Choreography e b)
  | CCall RecVar

newtype CDefSet e b = CDefSet [(RecVar, Choreography e b)]
newtype CProgram e b = CProgram (CDefSet e b, Choreography e b)
```

Fig. 2. Datatypes for choreographies in `hacc`.

(`CCond`) are based on Boolean expressions, and `CCall` invokes a named procedure.[1] `CProgram` is the type of choreographic programs, which includes definitions of named procedures (`CDefSet`) as well as the main choreography.[2]

Instead of working with our data types directly we define a convenient set of combinators for building choreographies. We provide a `prog` combinator that returns a `CProgram` given a pair of recursive procedure definitions and the main choreography. A choreography is given as a list of instructions (communications, selections of `CLeft` or `CRight`, conditionals and calls), all built using the corresponding combinators (`com`, `left`, `right`, `cond` and `call`, respectively), which are strung together to produce a CC term in our representation.

When compiling to Jolie, the programmer can configure the generated code by annotating interactions using the `ann` combinator. These annotations will be used to override the default names of the Jolie operations that implement them.

Example 1 (Distributed authentication). Figure 3 shows the distributed authentication choreography from [3] encoded as a `CProgram`.

Here, `Client` wishes to authenticate with `Ip` (identity provider) in order to receive a token from `Server`. `Client` sends its credentials to `Ip`, which checks them and communicates the result to both `Client` and `Server`. If the authentication was successful, `Server` sends a token to `Client`, otherwise the protocol terminates.

The terms of the local computation and Boolean expression languages are simply strings, which are included as-is into the generated executable code. ◁

[1] CC includes runtime terms, needed for the semantics. Programmers should not write them explicitly, so we do not include them in our datatype.

[2] In Coq, `DefSet` also includes the set of processes involved in each procedure. In theory, this set might not be computable, as there may be infinitely many procedures. This cannot happen in hand-written choreographies, so our wrapper computes this set.

```haskell
auth :: CProgram String String
auth = prog ([],
  [ann "authenticate" $ com c "credentials" ip credentials,
   cond ip "check@Util( credentials )"
     ([ann "authOk" $ left ip s, ann "authOk" $ left ip c,
       ann "acceptToken" $ com s "makeToken@Util()" c token],
      [ann "authFail" $ right ip s, ann "authFail" $ right ip c])])
  where [ip, s, c] = pids ["Ip", "Server", "Client"]
        [credentials, token] = vars ["credentials", "token"]
```

Fig. 3. The distributed authentication choreography in CC.

3 Code Generation

The projected behaviour of a process has type `Behaviour e b`, again parametric on the local languages (Figure 4). `Send` and `Recv` correspond to the respective

```haskell
data Behaviour e b
  = BEnd
  | Send Pid e Ann (Behaviour e b)
  | Recv Pid Var Ann (Behaviour e b)
  | Choose Pid Label Ann (Behaviour e b)
  | Offer Pid (Maybe (Ann, Behaviour e b)) (Maybe (Ann, Behaviour e b))
  | BCond b (Behaviour e b) (Behaviour e b)
  | BCall (RecVar, Pid)

newtype BDefSet e b = BDefSet [((RecVar, Pid), Behaviour e b)]
newtype Network e b = Network [(Pid, Behaviour e b)]
newtype BProgram e b = BProgram (BDefSet e b, Network e b)

epp :: CProgram e b -> Maybe (BProgram e b)
```

Fig. 4. Datatypes for processes in `hacc`.

communication actions, while `BCond` and `BCall` are as in CC. The remaining terms `Choose` and `Offer` are used to implement selections, where one process can *choose* which of the *offered* behaviours another process should execute. Note that a process does not have to offer behaviours for both labels.

Processes are named by identifiers and grouped into a `Network`, which is then paired with the projections of recursive procedures for each process (`BDefSet`) to form a `BProgram`. Finally, `epp` performs the projection of a `CProgram` to a `BProgram` using the extracted EPP as its foundation. We use the `Maybe` type to handle the case when a choreography is not projectable.

Example 2. Projecting the distributed authentication choreography from Example 1 gives us the SP program seen in Figure 5. Note how the annotations from the choreography have been preserved and propagated to the projection. ◁

```haskell
BProgram (BDefSet [], Network [
  (Pid "Client",
   Send (Pid "Ip") "credentials" (Ann "authenticate")
     (Offer (Pid "Ip")
       (Just (Ann "authOk",
              Recv (Pid "Server") (Var "token") (Ann "acceptToken") BEnd))
       (Just (Ann "authFail", BEnd)))),
  (Pid "Ip",
   Recv (Pid "Client") (Var "credentials") (Ann "authenticate")
     (BCond "check@Util( credentials )"
       (Choose (Pid "Server") CLeft (Ann "authOk")
         (Choose (Pid "Client") CLeft (Ann "authOk") BEnd))
       (Choose (Pid "Server") CRight (Ann "authFail")
         (Choose (Pid "Client") CRight (Ann "authFail") BEnd)))),
  (Pid "Server",
   Offer (Pid "Ip")
     (Just (Ann "authOk",
            Send (Pid "Client") "makeToken@Util()" (Ann "acceptToken") BEnd))
     (Just (Ann "authFail", BEnd)))])
```

Fig. 5. The projection of the choreography in Example 1.

Example 3. The SP program from Example 2 can now be compiled down to executable code (Figure 6). For each process, the backend generates a corresponding **service** block in Jolie. We omit the deployment configuration and show only the behaviour, whose structure follows that of the SP program.

Note how the strings used for the local computation and Boolean expressions (`makeToken@Util()` at `Client` and `check@Util( credentials )` at `Ip`) were incorporated into the generated code. Both make use of the `Util` module which we **embed** into all of the main services to provide a way for the user to supply local code that can be called out to if necessary – **embed** is Jolie for loading an internal service and is often used for encapsulating internal functions [10].

The annotations were used to specify the operation names exposed by the services (`authOk` and `acceptToken` for `Client`, `authenticate` for `Ip`, and `authOk` and `authFail` for `Server`). In general, they can specify arbitrary metadata that allows the programmer to control and guide the backend's code generation. ◁

Jolie

```
service Client {
  embed Util as Util
  ...
  main {
    authenticate@Ip( credentials )
    [ authOk() ] {
      acceptToken( token )
    }
    [ authFail() ] {
      nullProcess
    }
  }
}
```

```
service Ip {
  embed Util as Util
  ...
  main {
    authenticate( credentials )
    if( check@Util( credentials ) ) {
      authOk@Server()
      authOk@Client()
    } else {
      authFail@Server()
      authFail@Client()
    }
  }
}
```

```
service Server {
  embed Util as Util
  main {
    [ authOk() ] {
      acceptToken@Client( makeToken@Util() )
    }
    [ authFail() ] {
      nullProcess
    }
  }
}
```

Fig. 6. The executable Jolie code generated for `Client`, `Ip`, and `Server`.

4 Conclusion

We implemented a toolchain for compiling choreographies to executable code in Jolie. The complex step, computing the endpoint projection, is handled by certified code extracted from the Coq formalisation in [3]. This certified code is then combined with uncertified wrappers whose correctness is easy to check by hand. We illustrated the toolchain with a protocol for distributed authentication.

Acknowledgements This work was partially supported by Villum Fonden, grant no. 29518, and the Independent Research Fund Denmark, grant no. 0135-00219.

References

1. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: Giacobazzi, R., Cousot, R. (eds.) Procs. POPL. pp. 263–274. ACM (2013). `https://doi.org/10.1145/2429069.2429101`

2. Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. Theor. Comput. Sci. **802**, 38–66 (2020). https://doi.org/10.1016/j.tcs.2019.07.005, https://doi.org/10.1016/j.tcs.2019.07.005
3. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Certifying choreography compilation. In: Cerone, A., Ölveczky, P.C. (eds.) Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12819, pp. 115–133. Springer (2021). https://doi.org/10.1007/978-3-030-85315-0_8
4. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Formalising a turing-complete choreographic language in coq. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference). LIPIcs, vol. 193, pp. 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.15
5. Cruz-Filipe, L., Montesi, F., Peressotti, M.: A formal theory of choreographic programming. CoRR **abs/2209.01886** (2022). https://doi.org/10.48550/arXiv.2209.01886, https://doi.org/10.48550/arXiv.2209.01886
6. Hirsch, A.K., Garg, D.: Pirouette: higher-order typed functional choreographies. Proc. ACM Program. Lang. **6**(POPL), 1–27 (2022). https://doi.org/10.1145/3498684, https://doi.org/10.1145/3498684
7. Leesatapornwongsa, T., Lukman, J.F., Lu, S., Gunawi, H.S.: TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In: Proc. of ASPLOS. pp. 517–530 (2016)
8. Montesi, F.: Choreographic Programming. Ph.D. Thesis, IT University of Copenhagen (2013), http://www.fabriziomontesi.com/files/choreographic-programming.pdf
9. Montesi, F.: Introduction to Choreographies. Cambridge University Press (2023)
10. Montesi, F., Guidi, C., Zavattaro, G.: Service-oriented programming with jolie. In: Bouguettaya, A., Sheng, Q.Z., Daniel, F. (eds.) Web Services Foundations, pp. 81–107. Springer (2014). https://doi.org/10.1007/978-1-4614-7518-7_4, https://doi.org/10.1007/978-1-4614-7518-7_4
11. Myreen, M.O., Owens, S.: Proof-producing synthesis of ML from higher-order logic. In: International Conference on Functional Programming (ICFP). pp. 115–126. ACM Press (Sep 2012). https://doi.org/10.1145/2364527.2364545, https://cakeml.org/icfp12/index.html
12. Pohjola, J.Å., Gómez-Londoño, A., Shaker, J., Norrish, M.: Kalas: A verified, end-to-end compiler for a choreographic language. In: Andronick, J., de Moura, L. (eds.) 13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel. LIPIcs, vol. 237, pp. 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.27, https://doi.org/10.4230/LIPIcs.ITP.2022.27
13. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30:1–30:29 (2019). https://doi.org/10.1145/3290343, https://doi.org/10.1145/3290343