A Direct-Style Effect Notation for Sequential and Parallel Programs (Artifact)

David Richter
Technische Universität Damstadt, Germany

Timon Böhler
Technische Universität Damstadt, Germany

Pascal Weisenburger
Universität St. Gallen, Switzerland

Mira Mezini
Technische Universität Damstadt, Germany
hessian.AI, Darmstadt, Germany

Abstract

Modeling sequential and parallel composition of effectful computations has been investigated in a variety of languages for a long time. In particular, the popular do-notation provides a lightweight effect embedding for any instance of a monad. Idiom bracket notation, on the other hand, provides an embedding for applicatives. First, while monads force effects to be executed sequentially, ignoring potential for parallelism, applicatives do not support sequential effects. Composing sequential with parallel effects remains an open problem. This is even more of an issue as real programs consist of a combination of both sequential and parallel segments. Second, common notations do not support invoking effects in direct-style, instead forcing a rigid structure upon the code.

In this paper, we propose a mixed applicative/monadic notation that retains parallelism where possible, but allows sequentiality where necessary. We leverage a direct-style notation where sequentiality or parallelism is derived from the structure of the code. We provide a mechanisation of our effectful language in Coq and prove that our compilation approach retains the parallelism of the source program.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Software and its engineering → Concurrent programming structures; Software and its engineering → Parallel programming languages

Keywords and phrases do-notation, parallelism, concurrency, effects

Digital Object Identifier 10.4230/DARTS.9.2.17

Funding *David Richter*: German Federal Ministry of Education and Research *iBlockchain project* (BMBF No. 16KIS0902)
Timon Böhler: Hessian Ministry of Higher Education, Research, Science and the Arts (HMWK) via the project 3rd Wave of AI (3AI)
Pascal Weisenburger: The University of St. Gallen (IPF, No. 1031569); Swiss National Science Foundation (SNSF, No. 200429)
Mira Mezini: Hessian Ministry of Higher Education, Research, Science and the Arts (HMWK) via the project 3rd Wave of AI (3AI); German Federal Ministry of Education and Research *iBlockchain project* (BMBF No. 16KIS0902); German Federal Ministry of Education and Research and Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*

Related Article David Richter, Timon Böhler, Pascal Weisenburger, and Mira Mezini, "A Direct-Style Effect Notation for Sequential and Parallel Programs", in 37th European Conference on Object-Oriented Programming (ECOOP 2023), LIPIcs, Vol. 263, pp. 25:1–25:22, 2023.
`https://doi.org/10.4230/LIPIcs.ECOOP.2023.25`

Related Conference 37th European Conference on Object-Oriented Programming (ECOOP 2023), July 17–21, 2023, Seattle, Washington, United States

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2023 Call for Artifacts and the ACM Artifact Review and Badging Policy.

Dagstuhl Artifacts Series
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Scope

The artifact comprises a Docker image containing a Coq proof and a Scala program. The proof of the paper is mechanised in Coq. The compiler is implemented as well in Scala 3 via Macros, and a few tests.

2 Content

In the paper we have described the formal definition of a structurally recursive code-to-code translation function from a direct-style effect notation to monadic and applicative combinators, together with a proof of preservation of important properties. These can be found in the `coq` folder. Additionally, we mentioned an implementation in Scala, that works similar. These can be found in the `scala` folder.

More specifically the listings, figures, definitions, lemmas, and theorems of the paper correspond to the Coq source code in the following way:

- Listing 1 in the paper defines the class `Monad`, and the class `LawfulMonad`
- Listing 2 in the paper defines the inductive `ty`, the function `EVAL`, the inductive `ef`, the function `EF`, the inductive `tm`, the function `eval`, and the function `relabel`
- Listing 3 in the paper defines the function `PURE`, the function `AP`, and the function `JOIN`
- Figure 4 in the paper defines the function `SPAN` which corresponds to the use of `(fun _ => nat)` in the mechanisation, the function `WORK` which corresponds to the use of `(fun _ => nat)` in the mechanisation, the function `span`, and the function `work`
- Thereom 1 in the paper corresponds to the definition of the function `PURE` itself, e.g., the well-formedness of the translated term is guaranteed by the fact that PURE is well-typed.
- Lemma 2 "(AP respects semantics)" in the paper corresponds to the function `AP_eval` in the mechanisation
- Lemma 3 "(JOIN respects semantics)" in the paper corresponds to the function `JOIN_eval` in the mechanisation
- Lemma 4 "(relabel respects semantics)" in the paper corresponds to the functions `to_eval_src`, `to_eval_tgt` in the mechanisation
- Theorem 5 "(PURE preserves semantics)" in the paper corresponds to the function `eval_pres` in the mechanisation
- Theorem 5 "(PURE preserves semantics)" in the paper corresponds to the function `eval_pres` in the mechanisation
- Lemma 6 "(AP respects span and work)" in the paper corresponds to the functions `AP_span` and `AP_work` in the mechanisation
- Lemma 7 "(JOIN respects span and work)" in the paper corresponds to the functions `JOIN_span` and `JOIN_work` in the mechanisation
- Lemma 8 "(com is side-effect free)" in the paper corresponds to the functions `span_com_zero` and `work_com_zero` in the mechanisation
- Lemma 9 "(relabeled terms remain effect-free)" in the paper are separated into two steps, first the functions `to_span_src`, `to_span_tgt`, `to_work_src`, `to_work_tgt` in the mechanisation show that the span and work is preserved, and second the function `span_com_zero` and `work_com_zero` show that the span and work is not only preserved, but also equal to zero.
- Theorem 10 "(PURE preserves span and work)" in the paper corresponds to the functions `span_pres` and `work_pres` in the mechanisation

The Scala implementation provides a direct-style notation as an alternative to the for-comprehensions (do-notation), that compiles not only to sequential (monadic), but also parallel (applicative) combinators. This can be re-used by importing it. An example of how our artifact can be reused in new applications can be found in the the Readme inside the artifact.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at: `https://github.com/stg-tud/parseq-notation`.

4 Tested platforms

Hardware: There are no special hardware requirements. The device you execute the docker image should provide a performance comparable to a modern Computer or a Laptop. Software: We expect artifact reviewers to have preinstalled docker, a text editor, a terminal (tested with `bash`), and a .tar.gz extraction tool.

5 License

The artifact is available under Apache 2.0 License.

6 MD5 sum of the artifact

4a0db8605896be17ef789fc3bc4b7f59

7 Size of the artifact

1.00 GiB