Arc-Lang: Research Report

Klas Segeljakt
klasseg@kth.se

Frej Drejhammar
frej.drejhammar@ri.se

Paris Carbone
parisc@kth.se

Seif Haridi
haridi@kth.se

March 13, 2023

Abstract

Data analytics pipelines are becoming increasingly more complicated due to the growing number of requirements imposed by data science. Not only must data be processed and analyzed scalably with respect to its volume and velocity, but also intricately by involving many different data types. Arc-Lang is a programming language for data analytics that supports data parallel operations over multiple data types including data streams and data frames. In this paper we give a formal definition of Arc-Lang along with examples of its applications. We describe how Arc-Lang programs are translated into Rust using the MLIR compilation framework and then deployed on a distributed system. We compare Arc-Lang to different high-level DSLs for data intensive computing based on their problem domains, programming models, and implementations. Finally, we discuss future trends and open research questions in the area of DSLs for data intensive computing.

Contents

1 Introduction

The amount of data generated by the world is increasing at an alarming rate. Data is also becoming more accessible to the public domain. This has caused an emergence of applications in data science that must run in different environments that include consumer-grade hardware, large scale cloud centers, and resource-constrained edge devices. Streaming sensor data, graph social networks, relational tables of product information, and tensors of climate science image data are few examples of datasets that can scale to the point where they can no longer be managed by a single machine. As a result, distributed shared-nothing data-parallel systems have become the norm for data intensive computing. These systems are able to scale against increasing problem sizes by partitioning data and parallelising computation across machines. Distributed systems programming, in its barest form, is however known to be notoriously difficult.

Without proper abstraction, application developers must manage problems such as fault tolerance and coordination while considering tradeoffs in security and efficiency. To this end, distributed systems leverage high-level DSLs in the form of query languages, frameworks, and libraries, which are more friendly towards end-users. DSLs allow developers to focus on domain-specific problems, such as the development of algorithms, and to disregard engineering-related issues. Not only do DSLs lend themselves to improved ease of use, but also optimisation potential. DSLs in the form of intermediate languages have been adopted by multiple systems both as a solution to enable reuse by breaking the dependence between the program specification and its execution, and to enable target-independent optimisation.

Not only is data becoming more widespread, but also more complex. DSLs must be able to express algorithms over many different types of data. Examples of such DSLs include Keras[3] for machine learning and serving, SQL for data management, Apache Beam[1] for stream processing, and Cypher[2] for graph analytics. While these DSLs are highly optimised towards specific applications, users are met with problems when trying to combine fragments of different DSLs to solve more advanced problems. First, users must pay the price of serialisation and data movement costs between systems due to the lack of a common data format, and hardware resource contention due to the lack of a

common scheduler. Second, users must manage the impedance mismatch in guarantees between each system such as consistency, availability, and security. Third, users must learn how to program with tools of different systems, which may have widely different syntax and semantics.

We believe there is a need for a common DSL, IR, and execution environment for respectively expressing, optimising, and executing computations over different high-level data types. The functional requirements that this DSL must support are as follows:

- **Big Data Abstractions**: The DSL should be able to express algorithms over massive and rapidly growing collections of data. In particular, streams, tensors, frames, and graphs.
- **User-Defined Behavior**: The DSL should be able to express user-defined behavior. For example, user-defined functions and datatypes.

The non-functional requirements that the runtime system must satisfy are as follows:

- **Fault Tolerance**: The system should be able to sustain machine failure with efficiency. That is, without restarting the whole computation.

2 Arc-Lang

Arc-Lang is a language for data analytics.

2.1 Tour of Arc-Lang

This section highlights the main features of the Arc-Lang.

2.2 Features

3 Arc-MLIR

MLIR is used in the Arc-Lang compiler back-end. It is used for both standard compiler optimizations and transforms but also as convenient framework in which to express Arc-Lang-specific transforms. The MLIR/LLVM infrastructure for testing is also used to run Arc-Lang regression and unit tests.

The purpose of the Arc-Lang compiler back-end is to do domain specific transforms and optimizations which improves the efficiency of the program when running on Arcon. Doing transforms such as operator fusion and operator reordering requires a number of standard compiler techniques such as liveness analysis, constant propagation and common sub-expression elimination. By using the MLIR infrastructure, which implements many of these standard algorithms, we can concentrate on what is specific to Arc-Lang. By using MLIR we also have a robust and extensible intermediary representation with tools for

parsing, printing and verifying structure invariants of the processed program representation.

TODO: MLIR blurb and flesh out the bullets.

MLIR (https://mlir.llvm.org/) is a Multi-Level Intermediate Representation

- Extensibility
- Dialects
- Types
- Standard transforms and optimizations on custom dialects
- Tooling infrastructure: command line parsing, debug flags, pass ordering, error reporting.
- Testing support: Powerful DAG-matching tool to verify structure and syntax of output; Error report verification integration with the error reporting in the tooling infrastructure

3.1 Structure

The Arc-Lang front-end processes the Arc-Lang source code and produces a representation of the program in the arc-Lang MLIR dialect for further processing. The parts of the Arc-Lang compiler pipeline which uses MLIR is implemented in a tool called 'arc-mlir'. The tool is implemented using the MLIR tooling framework and allows the user to, on the command line, select which optimizations and transforms to run. Input to the 'arc-mlir' tool is MLIR-IR in the Arc-Lang dialect and output is in either: the Arc-Lang IR dialect, the Rust dialect or textual Rust source code.

3.2 The Arc MLIR Dialect

The Arc MLIR dialect is an MLIR dialect in which it is possible to represent all Arc-Lang language constructs in a way that allows the generation of a syntactically and semantically valid Rust program. The dialect consists of operations from the 'standard', 'scf', and 'arith' dialects provided by upstream MLIR, but also a number of custom operations and types specific to Arc-Lang.

3.2.1 Arc Dialect Types

The `arc` dialect includes a number of types which are not provided by one of the upstream dialects, these include:

- `arc.adt<string>` An opaque type which wraps a Rust type. It is preserved by all IR transformations. When Rust source code is output, values of this type will be declared as type `string`.
- `arc.enum` A Rust-style enum. A discriminated union where each named variant maps to a type. Structural equality applies to enum types.
- `arc.struct` An aggregate type which aggregates a set of named and typed fields. Structural equality applies to struct types.

- `arc.stream` A type which corresponds to event streams in Arc-Lang. The stream is instantiated with the type of the event it carries.

3.2.2 Custom Arc Dialect Operations

In the `arith` dialect, MLIR provides arithmetic operations on integer and floating point values. MLIR provides three integer types: one type which only specifies the number of bits `i<n>`, an explicitly signed integer type `si<n>`, and an explicitly unsigned integer type. The arithmetic operations on integers in `arith` are only specified for the `i<n>` integer type. In that, `arith` follows the model chosen by LLVM in that the signed/unsigned semantics for an operation is selected by the operation, for example `divi`/`divui` for signed/unsigned integer division. As both our input and output languages (Arc-Lang and Rust respectively) derive the signed/unsigned semantics from the type, we have chosen to use the explicitly signed/unsigned integer types. The alternative would require the component responsible for Rust output to derive the type of integer variables from the operations applied to them, something that is not always possible if no operations with different semantics are applied to them. Therefore the `arc` dialect defines its own polymorphic arithmetic operations operating on signed/unsigned integers.

TODO: operations

TODO: event handler

TODO: Structure of the Arc-Lang program: Each block produces a result, SSA-ish. No branches between blocks.

TODO: structured control flow

3.2.3 Rust MLIR Dialect

TODO: Operations which capture the structure of Rust.

TODO: Types are the rust type as a string.

TODO: Name mangling to produce Rust type names for the aggregate types.

TODO: Not intended to be the subject of any transforms or optimizations, that is done by rustc.

3.3 Standard Transforms and Optimizations

TODO: canonicalization; CSE; constant propagation and folding; constant lifting.

3.4 Custom Transforms

TODO: From SCF to BBs: In order to use additional optimizations and transforms.

TODO: From BBs to SCF, needed for Rust (no goto).

TODO: FSM-transform, for selective and nested receive in event handlers.

3.5 Rust Output

TODO: Abstracting away reference counting, borrows etc. Handled by macros in the runtime system libraries.

TODO: No formatting, rustfmt handles that.

3.6 Testing

TODO: Use Lit for unit and regression tests

TODO: Use built-in support in tooling to check that errors occur where we expect them.

4 Arc-Runtime

5 Related Work

6 Conclusion

References

[1] Apache Beam. Apache beam: An advanced unified programming model, 2017.

[2] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445, 2018.

[3] Keras Team. Keras: the python deep learning api, 2021.