Bachelor's Thesis

Towards a Formalization of Implicit Parallelism in Rust

Markus Walter

1. Reviewer	Prof. Dr.-Ing. Jerónimo Castrillón Chair for Compiler Construction TU Dresden
2. Reviewer	Dr.-Ing. Michael Roitzsch Barkhausen Institut
1. Supervisor	M.Sc. Andrés Goens Chair for Compiler Construction TU Dresden
2. Supervisor	Dr.-Ing. Sebastian Ertel Barkhausen Institut

TECHNISCHE UNIVERSITÄT DRESDEN

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Fakultät Informatik Institut für Technische Informatik, Professur Compilerbau

Task Description for Bachelors Thesis

for: **Markus Walter**

Major: Bachelor Informatik, 2018
Matriculation Nr.: 4767795

Title: **Formal Verification of Semantic Preservation in the Ohua compiler - A proof of concept**

Ohua is a parallelizing compiler that transforms sequential stateful applications into (micro-)service-based programs for micro-kernel-based operating systems and cloud infrastructures. The programming model of Ohua integrates into many existing imperative languages. At the same time, the internal representation of the compiler is the lambda calculus with transformations that provably preserve the semantics of the application to be parallelized.

So far the argument of the compiler to preserve the semantics of the compiled application is only limited and informal. First, the argument leaves out the second intermediate representation that essentially introduces the parallelism into the resulting program. Second, the argument has so far only been proven via a pencil-and-paper proof sketch that is disconnected to the current implementation of the compiler. In order to remove these short-comings, we target a formal verification of the whole compiler that is integrated into the development process. This will not only serve as certificate for the correctness of the current development state but makes sure that extensions do not break the compilers foundations.

This thesis shall focus on a first proof of concept that encompasses the transformation of the Rust subset into the the lambda calculus-based internal representation of the Ohua compiler. The implementation and the proof of semantic preservation of the transformation shall be carried out in Coq.

In particular, this Bachelors Thesis shall include the following tasks. The student shall:

1. Become familiar with the Coq theorem prover and the proof of semantic preservation.
2. Use existing definitions of the Rust programming language in Coq to define the subset supported by Ohua.
3. Define the Ohua's lambda calculus-based intermediate representation in Coq.
4. Define denotational semantics for both program representations.
5. Define a lowering from the Rust into the lambda calculus-based representation.
6. Prove (in Coq) that this lowering preserves semantics.
7. As a bonus: investigate how the defined abstractions can be integrated into the development process of the compiler using transpilers such as hs-to-coq.

Advisor: M.Sc. Andrés Goens, Dr.-Ing. Sebastian Ertel
1. Examiner: Prof. Dr.-Ing. Jeronimo Castrillon
2. Examiner: Dr.-Ing. Michael Roitzsch

Issued: 01.02.2021 Turn in by: 19.4.2021

Prof. Dr.-Ing. Jeronimo Castrillon

Dr.-Ing. Michael Roitzsch

Abstract

Ohua is a source-to-source compiler for multiple languages that aims to exploit parallelism by executing code that acts on separate state in parallel. The correctness of the transformation introducing parallelism has so far only been proven with a pencil-and-paper proof sketch. Because this proof is disconnected from the implementation, the argument of correctness is weak. In this thesis we take a new approach to proving semantic preservation by mechanizing a proof using the Coq proof assistant. This has two advantages. First, our proof is as correct and complete as our language models are. Second, it can be integrated in the Ohua development process and continuously adapted. We verify the transformation of a subset of Rust to Ohua's first intermediate representation. To this end, we first formally define the languages we transform from and to. We give semantic models for both, discussing trade-offs between different approaches to defining semantics. Then we show that the transformation preserves semantics for any correct Rust program.

Declaration

I hereby declare that I have written this thesis independently. I have acknowledged all sources I used for writing this thesis. To my best knowledge, this work is original and has never before been submitted by anybody at any university. I understand that any violation of this declaration may lead to the withdrawal of the attained degree.

Markus Walter
Dresden, April 19, 2021

Contents

Introduction 1

In the past two decades, computer architecture has shifted more and more towards multi-core processors [BC11]. In order to obtain performance gains from this architecture, programs have to be written for parallel execution [SL05]. Developing parallel applications comes with unique problems. Challenges arise e.g. in consistency and scheduling. Debugging such applications is more complicated than debugging sequential ones because of additional failure modes, such as deadlocks, race conditions and non-determinacy [Lu+08].

A variety of language constructs have been developed or proposed to tackle common challenges of parallel systems via automation. The programming models of those language constructs can be classified by their respective levels of abstraction. These levels range from threads, which are often a primitive of operating systems, to tasks, actors, dataflow operators and stateful functions. Abstractions are inherently opinionated, which means that by choosing which part to generalize they already make a decision which would have been made manually without the abstraction. This constitutes a trade-off between usability and freedom because more abstract programming models may restrict the programmer but enable quicker and safer implementation. For many applications highly abstract programming constructs suffice for implementation and eliminate certain error cases. However, there are applications for which a level fine-grained control is necessary that cannot be offered by high-level concurrency constructs.

This work focuses on Ohua, a source-to-source compiler that enables implicit concurrency. Using our classification, Ohua is on a high level of abstraction, stateful functions. The compiler, which is written in Haskell, currently supports subsets of Rust, Go, Java, Lisp and Ocaml [Wit20]. Internally, Ohua uses an intermediate representation (IR) that is based on the λ-calculus. Using this representation, a dataflow graph is constructed, on which semantics-preserving optimizations that introduce parallelism are performed. One advantage of Ohua is that no explicit language constructs or libraries for concurrency need to be used in the source program. Instead, the compiler handles all the parallelizing. This means that the source program can be written and debugged sequentially but executed in parallel [EFF15].

As in any compiler, errors in Ohua are extremely costly, since they may introduce bugs into correct source programs. Compiler bugs are usually hard to detect and have a broad impact since most programs need to be compiled. A lot of research has gone into minimizing compiler bugs, e.g. through automated testing, fuzzing and formal verification. Of those approaches, formal verification is the only one that can guarantee correct compilation [Yan+11].

Formally verifying the Ohua compiler would not only ensure the correctness of the transformation, which has so far only been proven via a pencil-and-paper proof sketch, but also guarantee that modifications of the compiler do not introduce any new bugs. In this thesis we give a proof of concept of how such a verification could be achieved.

The language subsets supported by Ohua have so far only been defined informally via the implementation. Going forward it would be useful to have formally specified subsets on which Ohua operates. This will not only be valuable for documentation but could provide a basis for tooling, such as a debugger. This thesis represents the first step towards having a specification of the supported subsets. By showing that the compiler's transformation is correct on the specified subset, we give a "lower bound" for a subset which the Ohua compiler can support.

1.1 Contributions

Verifying the entire compiler and integrating the proof into the development process is a very ambitious undertaking and beyond the scope of this thesis. We limit our verification to a small subset of Rust and Ohua's first IR as a proof of concept. In our transformation we make explicit use of Rust's ownership type system. We mechanize our proof using the Coq proof assistant [@21e]. Our proof of semantic preservation is not integrated into the compiler development.

For our proof we first define the syntax and operational semantics for subsets of Rust and Ohua's first IR. We then prove that the operational semantics for every program written in our Rust subset are equivalent to the operational semantics of the transformed program.

This thesis is structured as follows: in the second chapter we introduce the Coq proof assistant. We show how theorems are proved in Coq by giving a simple proof for an example language. In the third chapter we give an overview of Ohua and define our model of Ohua's intermediate representation, which we use in our proof. In the fourth chapter we explain Rust's ownership type system. We discuss existing semantic models for Rust, before specifying the syntax and semantics our supported Rust subset, μ_{Rust}. In the fifth chapter we outline the implementation of our transformation from Rust to Ohua and give an overview of our proof of semantic preservation. Further, we detail certain caveats of our proof and explain how these could be addressed. In the final chapter we give our conclusion on the viability of verifying the Ohua compiler. We finish by discussing avenues for future research, including more sophisticated semantic models and logics as well as how our models of Rust and Ohua could potentially be extended.

Defining semantics in Coq

2

In this chapter we give an introduction to the Coq proof assistant. We discuss usage options and trade-offs of formal verification. Then we define both syntax and operational semantics of an example language using Coq, introducing natural deduction with our operational semantics. Finally, we show Coq's proof system in action with a first proof of semantic preservation.

We chose to implement our proof in Coq for two reasons. First, Coq is very mature and has lots of tooling [@21e]. Second, Coq integrates with Haskell, the language Ohua is written in, via code generation [@21f; Spe+18]. This is important for future work, where we could target a direct integration of Coq with the main implementation.

2.1 A Coq primer

Coq is a proof assistant, which means that it can be used to prove the correctness of mathematical theorems and programs written in its own functional programming language, *Gallina*. Specifically, we prove invariants about the type system. It does so by transforming both the program's types and the proof into a small kernel of logical formulas which can be mechanically checked on consistency. The logical framework underpinning this is the Calculus of Inductive Constructions [BC04]. The fact that proofs in Coq are limited to the type system may seem confusing at first. After all, in most programming languages, the type system only captures a small part of the functionality. However, Coq's type system is very powerful, as it includes support for dependent types [Chl13]. This means that values can be encoded in types. Therefore, we can extend proofs to values.

Coq imposes the restriction that every program written in its language must terminate. This means that there are programs which can not be represented in Coq. However, many programs can be adapted to fit Coq's termination rules by slightly altering the implementation. Programs written in Coq can be translated into Ocaml, Haskell or Scheme, although more integrations exist [@21f].

In order to understand applications of proof assistants, it is worth comparing them to two other approaches of ensuring program correctness: unit tests and pen-and-paper proofs. Although they may seem similar at first, formal verification and unit tests are two very different approaches to tackling program correctness. Unit tests are easy to implement but can only guarantee that the correct output for *one specific input* is computed. In essence, they give a lower bound for correctness. Formal verification, on the other hand, is difficult

```
Inductive Op :=
  | AddOp
  | SubOp
  | EqOp
  | LeOp
  .

Inductive Lit :=
  | Bool : bool -> Lit
  | Int : Z -> Lit
  .

Inductive Ex :=
  | LitE : Lit -> Ex
  | VarE : string -> Ex
  | BinOpE : Op -> Ex -> Ex -> Ex
  | LetE : string -> Ex -> Ex -> Ex
  .
```

Listing 2.1: Syntax of *Ex*

to implement but can guarantee that the parts of the program we proved are indeed correct. Although Coq can verify the correctness of the proof, all proofs, except very simple ones, still have to be written by hand using Coq's proof language, *Vernac*. This is fundamentally different from a pen-and-paper proof, which can not rely on automatic checking. Instead, it has to be checked by other humans, leaving the possibility of error.

2.2 An example language

Ex, as in *Ex*ample or *Ex*pression, is a simple expression language. Its syntax definition in Coq is given in Listing 2.1. The syntax rules are defined in terms of Coq's inductive data types. Many functional programming languages, such as Haskell or ML, feature similar type constructors. The *Ex* language contains two data types, namely `Int` and `Bool`. To represent `Bool` values, it uses Coq's standard `bool` type and to represent `Int` values, it uses Coq's `Z` type, as implemented in the standard `BinInt` library. There exist four kinds of expressions in *Ex*: literal expressions, variable expressions, binary operations and let expressions. Some of the expressions use Coq's standard `string` type to represent variable names. Note that `LetE` takes two expressions, one for the value of the variable and one as a continuation with the updated environment.

```
Definition eval_pure_op (op : Op) (x y : Z) : Lit :=
  match op with
    | AddOp => Int (x + y)
    | SubOp => Int (x - y)
    | EqOp => Bool (x =? y)
    | LeOp => Bool (x <=? y)
  end.

Fixpoint eval_with_env (env : Map Lit) (ex : Ex) : option Lit :=
  match ex with
    | LitE l => Some l
    | VarE v => env v
    | BinOpE op v1 v2 =>
      match (eval_with_env env v1), (eval_with_env env v2) with
        | Some (Int a), Some (Int b) => Some (eval_pure_op op a b)
        | _, _ => None
      end
    | LetE v ex1 ex2 =>
      match eval_with_env env ex1 with
        | Some l => eval_with_env (add_item v l env) ex2
        | _ => None
      end
  end.

Definition eval (ex : Ex) : option Lit := eval_with_env (empty_map Lit).
```

Listing 2.2: Operational semantics of *Ex*

2.2.1 Operational semantics for *Ex*

We now want to define semantics for *Ex*. There are several approaches to doing so, of which we use the approach of defining operational semantics. In essence, operational semantics are given by showing a valid transformation of the source language to a formal target language. This can be achieved in Coq by implementing an interpreter. This allows us to reason about our programming language, because the interpreter itself is a formal Coq program. A simple interpreter implementation for *Ex* is shown in Listing 2.2.

The interpreter consists of three functions, `eval_pure_op`, `eval_with_env` and `eval`. We assume that `eval_pure_op` is self-explanatory to our readers. `eval` is a simple wrapper, which initializes `eval_with_env` with an empty environment. The actual interpretation of the expressions is carried out in the function `eval_with_env`. This function is called with two arguments: the root expression, to be evaluated, `ex`, and an environment, mapping variable identifiers (i.e. strings) to literal values. Our implementation of this data structure in Coq is given in Listing 2.3. It is essentially a stack of functions, which either yield the value, if the key matches, or delegate evaluation to the next function in the stack. The lowest function in the stack yields `None` for every key that has not

```
Definition Map (vt : Type) : Type := string -> option vt.

Definition empty_map (vt : Type) : Map vt := (fun _ => None).

Definition add_item {vt : Type} (k : string) (v : vt) (s : Map vt) : Map vt :=
    (fun (x : string) => if x =? k then Some v else s x).
```

Listing 2.3: Map definition

matched any entry in the rest of the stack. Note that instead of updating an entry, we can simply add a new one with the same key, since the map is ordered.

There is one case in which the evaluation can fail, which occurs when a variable is referenced that has not been previously declared. If the evaluation does not fail, it should return a single literal value, as all correct expressions can be reduced to one. Yet failure is an option, so the return value of `eval_with_env` cannot simply be `Lit`. Instead, it has to be wrapped in Coq's standard `option` data type. One interesting detail is that `eval_with_env` is declared as `Fixpoint`, whereas `eval` and `eval_pure_op` are declared as `Definition`. This is because of the three functions, only `eval_with_env` is recursive, an attribute that Coq requires to be explicitly annotated with the `Fixpoint` keyword.

2.2.2 *Ex*'s type system

We can formally define a type system for *Ex* using natural deduction [Gen35], given in Figure 2.1. Definitions in natural deduction are structured as a set of rules, each of which is usually annotated with a short name. The central construct of each rule is a horizontal bar, which has the same meaning as a mathematical implication ($\Longrightarrow$), where the top element implies the bottom one. Natural deduction is very flexible in terms of what can be written on both sides, as long as the implication holds. Note that implications in natural deduction can again be part of the condition or the result. Multiple conditions or results in a single rule are separated by a space, which corresponds to a mathematical *and* ($\wedge$). Optionally, an environment for rules can be given before the $\vdash$ symbol. The environment usually represents already computed values or types. In most cases the environment for the condition is the same as the environment for the result. If there is no condition for the result, the horizontal bar is omitted.

2.3 A first proof of semantic preservation

We define a simple transformation on our programming language, for which prove, that it does not alter the semantics of any program. To do so, we use Coq's proof system. We show that any transformed program evaluates to the same value as the non-transformed

$$\vdash \texttt{LitE (Int } i \texttt{)} : \mathit{Int} \qquad (\mathit{Int})$$

$$\vdash \texttt{LitE (Bool } b \texttt{)} : \mathit{Bool} \qquad (\mathit{Bool})$$

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \texttt{VarE } v : \tau} \qquad (\mathit{Var})$$

$$\frac{\Gamma \vdash ex_1 : \mathit{Int} \qquad \Gamma \vdash ex_1 : \mathit{Int}}{\Gamma \vdash \texttt{BinOpE AddOp } ex_1\ ex_2 : \mathit{Int}} \qquad (\mathit{Add})$$

$$\frac{\Gamma \vdash ex_1 : \mathit{Int} \qquad \Gamma \vdash ex_1 : \mathit{Int}}{\Gamma \vdash \texttt{BinOpE SubOp } ex_1\ ex_2 : \mathit{Int}} \qquad (\mathit{Sub})$$

$$\frac{\Gamma \vdash ex_1 : \mathit{Int} \qquad \Gamma \vdash ex_1 : \mathit{Int}}{\Gamma \vdash \texttt{BinOpE Eq } ex_1\ ex_2 : \mathit{Bool}} \qquad (\mathit{Eq})$$

$$\frac{\Gamma \vdash ex_1 : \mathit{Int} \qquad \Gamma \vdash ex_1 : \mathit{Int}}{\Gamma \vdash \texttt{BinOpE Le } ex_1\ ex_2 : \mathit{Bool}} \qquad (\mathit{Le})$$

$$\frac{\Gamma \vdash ex_1 : \tau_1 \qquad \Gamma \cup \{ex_1 : \tau_1\} \vdash ex_2 : \tau_2}{\Gamma \vdash \texttt{LetE } v\ ex_1\ ex_2 : \tau_2} \qquad (\mathit{Let})$$

Fig. 2.1: Type system of *Ex*

```
Fixpoint const_fold (ex : Ex) : Ex :=
  match ex with
    | BinOpE op (LitE (Int l1)) (LitE (Int l2)) => LitE (eval_pure_op op l1 l2)
    | LetE v ex1 ex2 => LetE v (const_fold ex1) (const_fold ex2)
    | _ => ex
  end.
```

Listing 2.4: Implementation of constant folding for *Ex*

program. This serves as a model for our proof of semantic preservation in Ohua, which is essentially constructed the same way.

The transformation we want to prove semantic preservation for is simple constant folding. Its implementation is given in Listing 2.4. The implementation checks for binary operations operating on two literal integer values. On any such pattern it computes the resulting value. In order to find all matching binary operations, it recursively searches sub-expressions of `LetE` expressions. Note that we could have similarly matched sub-expressions of `BinOpE` and checked if they can be transformed into integers. This would have had the effect of propagating results that have been folded. Our transformation is only intended as an academic example to showcase Coq's proof system. We chose not to include propagating folds, as this would have unnecessarily complicated our proof, which we outline now.

Proofs in Coq are composed of two parts: the Proposition, i.e. *what* we prove, and the actual proof, i.e. *how* we prove it. Our proposition is going to be the following:

```
Lemma const_fold_correct_env : forall (ex : Ex) (m : Map Lit),
  eval_with_env m (const_fold ex) = eval_with_env m ex.
```

We want to show that for all expressions `ex` and for all environments `m`, evaluating `ex` with `m` yields the same result as evaluating `const_fold ex` with `m`. We enter proof mode by writing:

```
Proof.
```

Proofs are realized by operating on given *assumptions* to produce a *goal*. Both are sets of conjunctive terms. We show that our goals are implied by our assumptions. To do so we apply sound transformations, called *tactics*, which operate on our assumptions and goals. Tactics can introduce, modify and consume assumptions and goals. If all goals have been consumed, the proof is complete. The usage of tactics guarantees, that the implication of the "old" goals by the "old" assumptions is implied by the implication of the "new" goals by the "new" assumptions. This principle is illustrated in Figure 2.2.

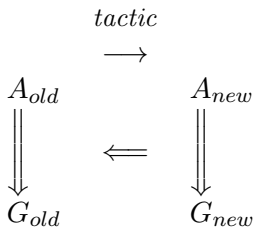


Fig. 2.2: Application of a tactic

We start our proof with empty assumptions the single stated goal:

```
forall (ex : Ex) (m : Map Lit),
  eval_with_env m (const_fold ex) = eval_with_env m ex.
```

Our first assumption is `ex : Ex`, which essentially corresponds to the mathematical definition "*let* ex *be of type* Ex". It is introduced by applying the tactic:

```
intro ex.
```

Because the `Ex` data type is defined inductively, our proof has to account for that property. We initiate a structural induction by applying the tactic:

```
induction ex.
```

our original goal is consumed and four new goals are produced, one for each data constructor of `Ex`. It also automatically produces inductions hypotheses as assumptions. The base cases of `LetE` and `VarE` can each be handled using the tactic:

```
reflexivity.
```

This automatically expands definitions and consumes goals of the kind "$x = x$". Next up is `BinOpE`. We are in luck, because `const_fold` does not operate on recursively defined binary operations. This means that we simply have to do case analyses on both binary arguments:

```
destruct ex1; destruct ex2; try destruct l; try destruct l0;
reflexivity.
```

The `destruct` tactic does just that: it produces a new goal for each possible data constructor of a variable and consumes all previous goals containing this variable. We destruct the variables `ex1`, `ex2`, `l` and `l0`, which have been automatically introduced by previous tactics and represent the binary arguments. In every case we can use `reflexivity` to easily show that semantics are preserved.

Finally, only `LetE` remains. This is where we make use of our inductions hypotheses. We are trying to prove the term:

```
forall m : Map Lit,
  eval_with_env m (const_fold (LetE s ex1 ex2)) =
  eval_with_env m (LetE s ex1 ex2)
```

We use

```
intro m0.
```

to introduce the variable `m0` and

```
simpl. rewrite IHex1.
```

to apply the fist induction hypothesis. Now we still have to prove the following term:

```
match eval_with_env m0 ex1 with
  | Some l => eval_with_env (add_item s l m0) (const_fold ex2)
  | None => None
end =
match eval_with_env m0 ex1 with
  | Some l => eval_with_env (add_item s l m0) ex2
  | None => None
end
```

This can easily be achieved by applying our induction hypothesis:

```
forall m : Map Lit,
  eval_with_env m (const_fold ex2) =
  eval_with_env m ex2
```

The complete proof is given in Listing 2.5. This particular proof only covers the function `eval_with_env`, but we can trivially extend it to work with `eval`:

```
Theorem const_fold_correct : forall (ex : Ex),
  eval (const_fold ex) = eval ex.
Proof.
  intros. apply const_fold_correct_env.
Qed.
```

Because `const_fold_correct_env` is defined on all possible environments `m`, it can be applied to the environment `empty_map Lit`, which is used in the `eval` Function.

```
Lemma const_fold_correct_env : forall (ex : Ex) (m : Map Lit),
  eval_with_env m (const_fold ex) = eval_with_env m ex.
Proof.
  intro ex.
  induction ex; try reflexivity.
  - destruct ex1; destruct ex2;
    try destruct l; try destruct l0;
    reflexivity.
  - intro m0. simpl.
    rewrite IHex1.
    destruct (eval_with_env m0 ex1).
    + rewrite IHex2. reflexivity.
    + reflexivity.
Qed.
```

Listing 2.5: Proof of semantic preservation of constant folding in *Ex*

The Ohua framework 3

In this chapter we introduce Ohua. We first give an overview of the programming model used by the Ohua framework. We illustrate how Ohua introduces parallelism to sequential programs. Then we introduce the syntax and semantics of the language modeled after Ohua's first IR, which we use for our proof. We call this language the λ-IR.

3.1 Implicit parallel programming

Ohua is a framework for writing parallel applications that don't require explicit use of parallel language constructs. This kind of parallelism is called implicit parallelism. Ohua consists of a compiler and runtime. The Ohua compiler is language independent, but compiler bindings and the runtime are adapted for each supported language. The program being parallelized is called *algorithm*. It can call into library functions that are not effected by Ohua's parallelization. These library functions are not restricted by Ohua's programming model. Algorithms used to be written in separate files, in a language that resembles Lisp or OCaml. The compiler has in recent versions been adapted to work with Rust and Go, where Algorithms are written directly in the respective host language [Wit20]. In this thesis we focus on Rust, as this integration is now the main development focus of the Ohua project.

Ohua internally distinguishes between two kinds of functions. Functions that operate on state, i.e. mutable input data, are called *stateful functions*. Functions which operate on immutable data are called *pure* or *stateless functions*. Programs in Ohua are first transformed into an expression IR, which is loosely based on the λ-calculus and then lowered into a dataflow graph representation. Nodes in the dataflow graph represent function calls or helper nodes with control flow information for the runtime. Edges indicate data flow between function calls. Ohua achieves parallelism by executing independent nodes in the dataflow graph, i.e. nodes that do not share state, in parallel. This form of parallelism is called *task parallelism* [Ert+19b]. The dataflow graph is executed by the Ohua runtime, which dynamically schedules the nodes [Ada16] [Ada19]. In a compiled language such as Rust execution with the runtime would correspond to linking against the runtime and compiling both using a Rust Compiler. This process is visualized in Figure 3.1. The semantic preservation of Ohua's compiler and runtime execution has been proven on paper for a λ-calculus-like language [Ert+19a]

To safely parallelize the program, Ohua's programming model imposes the following restrictions on data-flow graphs. First, no state may be shared between independent

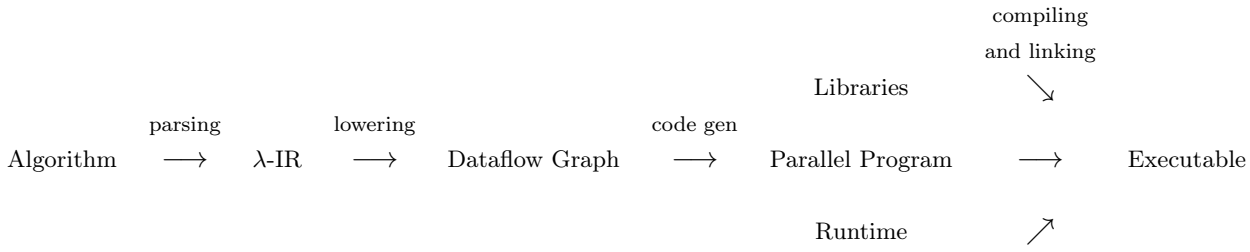


Fig. 3.1: Compilation stages of Ohua

nodes in the dataflow graph. This implies that mutable global variables are not allowed. All function calls that operate on one state have to be dependent on each other. This property rules out the possibility of arising deadlocks or race conditions and is already enforced through Rust's ownership type system, which we introduce in section 4.1. State has to be passed explicitly though arguments and return values. This means that mutable global variables or closures are not allowed. Immutable data can be shared across independent nodes. Our current implementation of Ohua for Rust works with a restricted version of stateful functions in which only mutable `structs` are recognized as state which can be parallelized on, other mutable data is not allowed within the algorithm part of the program. Each stateful function can only operate on one state, though this is a restriction that could be relaxed in future versions of the compiler [Ada19]. These restrictions are visualized in Figure 3.2, where s represents a single state, im represents immutable data and f represents a simple stateful function.

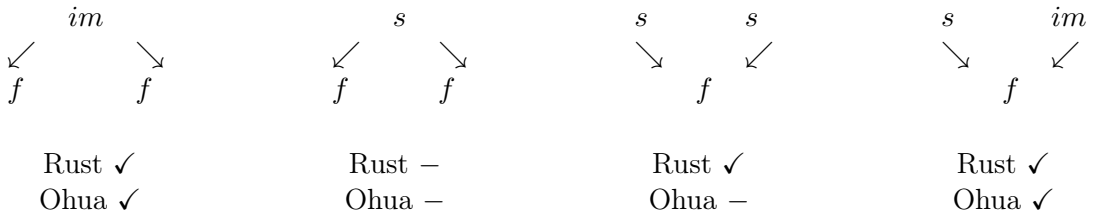
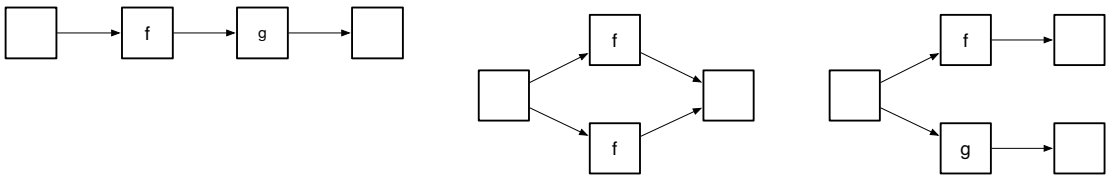


Fig. 3.2: Dataflow restrictions in Rust and Ohua

3.1.1 smap

Ohua provides another tool for achieving implicit parallel programming: `smap`. `smap` can be thought of as a combination of the functional *map* and *fold* or *reduce* operations. It folds input state over an input vector and produces both a new state and a modified output vector. This construct is general enough to be applied to both functional and imperative programming patterns. The corresponding imperative programming construct would be a loop over the input vector, where each iteration operates on an input vector element and the input state.

`smap` exploits two patterns for parallelization: *data parallelism* and *pipeline parallelism*. Data parallelism is achieved when the input state is not modified in the computation, i.e. when `smap` is used as a simple *map* operation. In this case all elements of the input vector are mapped over in parallel. Pipeline parallelism is achieved when `smap` operations with different states are chained on one input vector [Ada19]. In this case the computation can be carried out in pipeline fashion. The three types of parallelism are shown in Figure 3.3 [Ert+19b].



(a) pipeline parallelism (b) data parallelism (c) task parallelism

Fig. 3.3: Types of parallelism in Ohua

To show Ohua's parallelization opportunities in action, consider the Rust algorithm in Listing 3.1 as an example. Pipeline, data and task parallelism can be achieved here. All three iterations are represented as `smap` operations in Ohua. The first iteration over `l` corresponds to a pure *map* operation, so it can be executed in a data-parallel manner. The other two iterations, operating on the separate states `c1` and `c2` but on the same data, are executed with pipeline parallelism. The final two statements can be executed in a task parallel manner, since they are state independent.

The corresponding dataflow graph is shown in Figure 3.4. Control flow dependencies are dotted edges, state dependencies are dashed ones. `smap` and `collect` are helper nodes to initialize and exit iterations. The type of parallelism is indicated by a nodes' color. Yellow represents data parallelism, green represents pipeline parallelism and blue represents task parallelism.

Ohua is at the time of writing being rewritten to improve error handling and adapt the programming model to microservices [Ert+18]. The λ-IR, though general and language-independent, has proven insufficient for error handling. This is because a lot of information needed for good error behavior, such as type information, is being evicted while lowering the host language into the λ-IR. Enriching the IRs with more information means losing some generality. With the rewrite it will become harder to adapt Ohua for new host languages. With Rust being the language that Ohua is best adapted to and one of the premier languages for microservices [@21c], the rewrite currently focuses on Rust as a host language. Abandoning the λ-IR means that we lose our proof of semantic

```rust
let mut c1 = Counter::new(); // 0
let mut c2 = Counter::new(); // 0
let mut l = vec![1, 2, 3];

let mut l1 = Vec::new();
for i in l {
  l1.push(i + 1);
};
l = l1;

let mut l1 = Vec::new();
for i in l {
  c1.increment(i);
  l1.push(i + 1);
};
l = l1;

for i in l {
  c2.increment(i);
};

c1.get_value(); // 9
c2.get_value(); // 12
```

Listing 3.1: Example algorithm in Rust

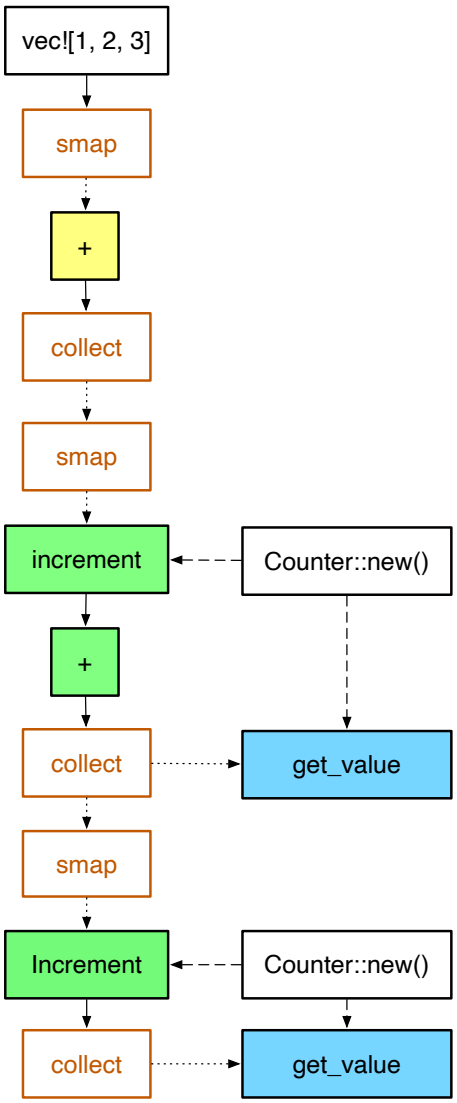


Fig. 3.4: Dataflow graph of the example algorithm

Functions:

Fun	::=	`PureFun` *string* (*list string*)	pure function with function identifier and parameter identifiers
	|	`STFun` *string string* (*list string*)	stateful function with function identifier, state identifier and parameter identifiers

Expressions:

Expr	::=	`Lit` *value*	literal value of the host language
	|	`Var` *string*	
	|	`Let` *string Expr Expr*	
	|	`FunApp` *Fun* (*list Expr*)	function application with list of arguments
	|	`Lambda` *string Expr*	a lambda abstraction
	|	`App` *Expr Expr*	a lambda application
	|	`Smap` *string Expr Expr*	a `smap` construct
	|	`Cond` *Expr Expr Expr*	an if expression

Fig. 3.5: Grammar of the λ-IR

preservation. This thesis is partly motivated by that consequence, as a new proof of semantic preservation is needed and the type-based nature of the rewritten Ohua compiler aligns itself well with the type system of proof assistants such as Coq.

3.2 Introducing the λ-IR

In Figure 3.5 we give the grammar of the λ-IR. It is defined in a BNF-like notation, as are all other grammars we define in this thesis. The grammar gives rules for two non-terminal symbols, *Fun* and *Expr*. Every program of the λ-IR is an expression. This means that *Expr* represents a node in the λ-IR AST. *Fun* represents a function in the source language. The main point of the λ-IR is that opportunities for parallelism are made explicit, i.e. they have already been determined in the transformation to the IR.

This explicitness can be observed in the two function types, `PureFun` and `STFun`. `PureFun` is used to represent pure functions, i.e. functions that only operate on immutable data. It is composed of a function name and a list of identifiers of parameters. `STFun` represents a simple state thread, i.e. a function that can operate on one mutable state. In addition to the function name and pure parameters it has one data field for a state identifier. In a previous version of this proof, functions would be inlined. This was done by making the function body of the host language part of the `STFun` or `PureFun` data type. However, the function representation which simply uses identifiers is easier to prove correct. This is because our Rust subset also has function call semantics using identifiers, as discussed in section 4.4.

The grammar is generic, meaning that no language constructs of one host language are part of it. It needs to be generic because Ohua aims to integrate with more than on host language. Code of the host language can be embedded in the λ-IR using `Lit`. This embedded code is not operated on by Ohua, only the statements that have been lowered into the λ-IR are. In the following paragraphs we describe the meaning of the different data constructors for *Expr*. Keep in mind that Ohua itself assigns no values to the embedded code of the host language and, therefore, does not evaluate any expression.

`Var` represents a reference to a value of the host language. Note that this value could be located both in the global environment and in the local environment. The global environment is known at compile time and consists of the definitions which the algorithm operates on. The local environment is constructed at run time from variables introduced by `Let`.

Any `Let` expression consists of three parts: a variable identifier, a value expression and a continuation expression. The value expression is evaluated to a value of the host language and bound to the identifier in the local environment. Then the continuation expression evaluated with the new local environment. `Let` has another use: by giving a variable identifier that is never referenced in the continuation, we can simulate statements with no return value.

`FunApp` represents a function call. Its constructor takes a `STFun` or a `PureFun` and argument expressions as parameters.

The `Lambda` expression creates a new lambda abstraction, i.e. an anonymous function with one parameter. Lambda expressions are used in combination with `App` or `Smap` expressions. `App` applies a given `Lambda` expression to an argument. It evaluates the argument, binds it to the lambda parameter and evaluates the lambda expression body. Note that a `Lambda` and a `App` expression in combination have the same effect as a `Let` expression, as both simply introduce a new variable for a continuation. Lambda expressions can be nested. When the body of a lambda expression is another lambda expression, then is can be seen as an anonymous function with two parameters.

The `Smap` expression represents an `smap` invocation. This has been described in detail in section 3.1.1. It takes three arguments, the first being the identifier of the state it acts on, the second being a lambda abstraction, which is folded over the third argument, a sequence of the host language, embedded via a `Lit` expression.

`Cond` represents an if-condition. The first argument evaluates to a boolean of the host language. If this result is true, then the second expression argument is evaluated, else the third one is. This data constructor is only an annotation to construct Ohua's dataflow graph. Recall that none of the expressions are evaluated in Ohua, they are only transformed to introduce parallelism. However, they are evaluated with in the operational semantics we need to define for our proof of semantic preservation.

3.3 Semantics for the λ-IR

Having loosely described the semantics of our λ-IR in the previous section, we are now going to give a detailed view of the implementation of the operational semantics in Coq-like pseudocode. We define semantics for `Lit`, `Var`, `Let`, `FunApp`, but not for `Lambda`, `App`, `Smap` or `Cond` because our Rust subset is too small to allow any meaningful applications of these constructs.

Our evaluation function is parameterized with a global environment and a local environment. These map identifiers to values of the host language, in our case Rust. The global environment does not change during evaluation, while the local environment is empty at first and is built up during the evaluation. The environments are implemented using the same data structure as *Ex*'s and Rust's evaluation functions, which is introduced in section 2.2.1.

Furthermore, the evaluation function for the λ-IR takes an evaluation function for the host language as an argument. This is because the λ-IR is generic, but the operational semantics need to evaluate any program in the λ-IR to a value of the host language. The evaluation function for our Rust subset, which we define in section 4.4, also takes these environment arguments, but needs additional ones for type checking. For the λ-IR, type checking has already been performed during the transformation from Rust. This is covered in detail in section 5.

Any `Lit` expression can simply be evaluated by applying the Rust evaluation function to the embedded Rust value. This is illustrated in Listing 3.2.

```
lambda_eval global_env local_env rust_eval (Lit value) :=
  rust_eval global_env local_env value.
```

Listing 3.2: Evaluation of a literal value

`Var` expressions are evaluated by looking up the value of the variable in the local and global environments. This is shown in Listing 3.3. Note that the local environment has a higher precedence than the global one, so we attempt to look up the value in the local environment first. The function's return type is an `option`, or `Maybe` in Haskell's terminology, to represent a possible failure of evaluation. This has been excluded from the other evaluation definitions for readability. If the variable is not present in our environments, the function returns `None`.

Listing 3.4 gives the evaluation of `Let` expressions. As described in the previous section, the evaluated value expression is added to the local environment before the continuation is evaluated.

Finally, let's look at the operational semantics of function calls. Function calls in the λ-IR are simply mapped back to Rust's function calls. This is because of the model of

```
lambda_eval global_env local_env rust_eval (Var ident) :=
  match local_env ident with
  | Some value => value
  | None => global_env ident
  end.
```

Listing 3.3: Evaluation of a variable

```
lambda_eval global_env local_env rust_eval (Let ident expr cont) :=
  let value := lambda_eval global_env local_env rust_eval expr
  in lambda_eval global_env (add_item ident value local_env) rust_eval cont.
```

Listing 3.4: Evaluation of a `Let` binding

function calls, which we use in our λ-IR as well as in our Rust subset. Here function calls which are represented by simply giving the function identifier and argument expressions, as explained in section 4.4.

Pure functions are mapped to Rust's pure functions. State threads are transformed to the corresponding method calls, as both operate on mutable state. As described in section 4.3.1, the first argument of a state thread call is the mutable state it operates on. This is why when evaluating the `FunApp` of an `STFun`, we have to deconstruct the argument list.

Normally a semantic model for function calls would have to do a type check. We would have to check if the argument types match the parameter types. For the state thread we would have to additionally check if the type of the first argument, our state, matches the state type in the `STFun` data constructor. This type check is already performed during the transformation from Rust to the λ-IR, which is described in section 5.

Here we can see that our specific implementation of operational semantics is not entirely generic, since we directly return a Rust value. Because our evaluation function returns Rust expressions and function calls in Rust are not evaluated to a more fine-grained degree than giving the function identifier and arguments, we have to use a Rust value directly. If we evaluated function bodies, and thus gave a more fine-grained operational semantics, our evaluation function could be generic, since we could use `rust_eval` directly on the function body.

```
lambda_eval global_env local_env rust_eval
  (FunApp (PureFun ident params) args) :=
  rust_eval global_env local_env (Rust.Call ident args).

lambda_eval global_env local_env rust_eval
  (FunApp (STFun ident state params) args) :=
  match args with
  | s :: args' => rust_eval global_env local_env (Rust.MethodCall ident s args')
  | _ => Rust.Err
  end.
```

Listing 3.5: Evaluation of a function call

The Rust language

4

In this chapter we give an overview of the Rust language and its type system. We discuss existing formal semantic models of Rust. Then we define the syntax of a Rust-like language, which we use for our proof and which we call μ_{Rust}. We discuss why certain language features have been included in μ_{Rust}. Having defined the syntax we give a specification of its operational semantics.

4.1 A tour of Rust

Rust is a systems programming language which aims to offer safety and control at the same time. For almost all widely used languages, programmers have to choose between the two. Take memory management as an example. In Java memory is managed via garbage collection. This offers safety, as memory leaks are not as common as in C++ and no dangling references can exist. At the same time, the programmer has less control over the memory layout that they would when using a language with manual memory management such as C++. For example, in Java all non-primitive values are boxed, so it is hard to optimize compound data for cache efficiency. Furthermore, it is hard to predict when and for how long Java's garbage collector will run. In C++ the programmer has control over these issues. They can freely choose a memory layout for their data and they can choose when to allocate and deallocate data. However, this level of control comes at the cost of safety. The unchecked use of interior pointers, i.e. pointers that point into a data structure, can lead to dangling pointers and use-after-free errors. As for manual memory management, it is not uncommon to have memory leaks if data is incorrectly deallocated. Rust offers both the safety of garbage collection and the control of manual memory management through its system of compile-time checked automatic deallocation. This is possible because Rust enforce the correct use of pointers in its type system.

Types in Rust do not just annotate the class of values a variable can hold, they also show a property called *ownership*. Ownership is the central concept for ensuring correct use of references in Rust. The underlying idea of Rust's ownership system is that values may either have a single mutable reference or multiple immutable references, but not both at the same time. This concept is sometimes called *mutability XOR aliasing*. This prevents data races as no shared state is permitted between threads, a feature that is also the central requirement to Ohua's parallelization. Here we can already see that the restrictions enforced by Rust resemble the ones enforced by Ohua [Jun20].

To illustrate this principle, we give the program in Listing 4.1. Examining the use of the variable `x`, we can see Rust's borrow semantics in action. There can exist either multiple read-only references of `x`, as shown in the function call of `add`, or one mutable reference, as shown in the call of `inc`. If we had attempted to call `inc` with two mutable references of `x` then Rust's borrow checker would have rejected our code, even if it would not have led to a run time conflict in this case.

```
fn add(a: &i32, b: &i32) -> i32 { *a + *b }
fn inc(a: &mut i32, b: &mut i32) { *a = *a + *b }
let mut x: i32 = 1 + 2;
let mut y: i32 = add(&x, &x);
inc(&mut x, &mut y);
```

Listing 4.1: Mutable and immutable references

Let's look at how ownership works in practice. Having a plain variable `v` means you own it fully. `v` is called the *owner* of its value. When the owner goes out of scope, its lifetime ends and its value is automatically deallocated. When copying `v` a distinct owner for the new value with no relation to `v` is created. This process is called *cloning*. This happens for example when passing stack-allocated values as arguments, returning them from functions or assigning them to a new variable. Heap-allocated values can be cloned using a clone function [Jun+21].

When heap-allocated values are passed as arguments, returned from functions or assigned to a new variable without being explicitly cloned, the are *moved*. This is a process in which the full ownership is transferred to another variable. The original owner is invalidated. Consider the example given in Listing 4.2. `v` is moved into `a` when calling `f`. This means that after calling `f`, `v` can no longer be used to refer to the Vector. If it were not for that rule, `v[0]` would be an invalid memory access, because `a` and, therefore, `v` get deallocated with the use of `drop` in `f` [Jun+21].

```
fn f(a: Vec<i32>) {
  drop(a);
}
let v = vec![1, 2, 3];
f(v);
v[0]; // error: borrow of moved value: `v`
```

Listing 4.2: Access after move

Moving heap-allocated values every time they are passed to or from a function is quite cumbersome. This is why ownership can be temporarily transferred to a different owner. This is called *borrowing*. "Temporarily" in this contexts means that the lifetime of the borrowing owner must not exceed the lifetime of the original owner. If the original owner is used, the lifetime of the borrowing owner expires. In other words, the original owner may not be used while the borrowing owner exists [Jun20]. Two kinds of borrowing exist

in Rust: mutable and immutable borrows. At any point in the program a value can either have at most one mutable borrow or multiple immutable borrows, but not both. In Listing 4.3 we can observe the restriction that the original owner may not be moved while the borrowing owner's lifetime is active. This again prevents an invalid memory access [Jun+21].

```
fn f(a: &mut Vec<i32>) {
  drop(*a); // error: cannot move out of `*a` which is behind a mutable reference
}
let mut v = vec![1, 2, 3];
f(&mut v);
v[0];
```

Listing 4.3: Move of a borrowed value

Borrow restrictions are not only useful in multi-threaded environments, where they control data accesses. They also eliminate errors related to interior pointers in single-threaded environments. Examples for such errors are iterator invalidation and use-after-free. In Listing 4.4, we see a use-after-free bug. The bug occurs because `push_back` reallocate the buffer for `v`. In this scenario, `vptr` would not hold the intended value anymore [Jun+21].

```
std::vector<int> v {1, 2};
int *vptr = &v[1];
v.push_back(3);
*vptr; // bug: use after free
```

Listing 4.4: Use after free bug in C++

This invalid memory access is prevented by Rust, as shown in Listing 4.5. `vptr`, which is a mutable borrow, as indicated by `&mut`, is used before and after `push`. `vptr`'s lifetime is active on `push`. We attempt to create a second mutable borrow of `v` as an argument for `push`, which leads to an error.

```
let mut v = vec![1, 2];
let vptr = &mut v[1];
v.push(3);
*vptr; // error: cannot borrow `v` as mutable more than once at a time
```

Listing 4.5: Use after move error

The borrow restrictions can be disabled in unsafe blocks. This is discouraged outside of standard library modules that absolutely need such behavior, e.g. Mutex. These modules can be integrated in safe code without restrictions. The verification of unsafe libraries and its inclusion in safe code is an ongoing area of research [Jun+19]. Ohua does not allow unsafe code.

4.2 Formalized models of Rust

The formal verification of Rust is an ongoing effort coordinated by the formal verification working group [@18]. For our proof to work as outlined in chapter 5, we need to have operational semantics of Rust implemented in Coq. In this section we cover proposed formal semantic models of Rust and evaluate their use for our proof of semantic verification. In order to be a valid option for our proof, a semantic model would have to fulfill three requirements: it would have to work with a source-level representation of Rust, it would have to implement the entire subset of Rust supported by Ohua and it would have to be implemented in a proof assistant.

Patina [Ree15] is a formal semantics with a special focus on Rust's ownership system. It was for an early (pre 1.0) version of Rust. As such it's ownership semantics are outdated at this point. Furthermore, no implementation is available, since Patina is defined in purely syntactic terms [Wei+20].

KRust [Wan+18] and *RustSem* [Kan+20] are independent projects with define executable operational semantics for Rust in the K framework [@21b]. RustSem was previously called K-Rust, but is not related to KRust. Both define semantics for source-level Rust and implement a subset large enough for our needs. Unfortunately, the K framework does not integrate Coq, or with any proof assistant for that matter. This rules it out for use in our proof.

Oxide [Wei+20] gives formally semantics for a language close to source-level Rust. Although it is a simplified version of Rust, for example it does not implement `traits`, it covers the subset supported by Ohua. Oxide is currently being implemented in OCaml, which means that it could be integrated with our proof via *coq-of-ocaml* [@21a]. However, so far only the type checker has been implemented [@21d]. This is insufficient for our proof as we need full operational semantics.

RustBelt [Jun+17] gives a complete semantic specification for an intermediate representation of the Rust compiler, the Mid-level Intermediate Representation (MIR), is given. In MIR many high-level syntactic structures have already been lowered to a continuation passing representation. RustBelt is implemented in Coq, which means that it could be seamlessly integrated with our proof. Rust's semantics are specified through a transformation from Rust to a core language. The transformation is verified with a separation logic.

For our purposes RustBelt does not fit the required criteria since we need a Rust subset that is aligned with Rust's AST and not MIR. Many high-level constructs that are critical for Ohua to function effectively have already been decomposed in MIR. For example, loops, which are used in Ohua to create parallelism via `smap`, have already been lowered to continuations in MIR [Jun+17].

None of these semantic models fulfill all our requirements. Interestingly, none of these models attempt to cover all of Rust, for example the trait system is absent in all of them. It is still subject of research, if and how the trait system can even be modeled. This means that there is a possibility that the Rust language may never be fully formalized. However, for Ohua's algorithms we only need a small subset, for which a formalization seems attainable.

4.3 Choosing a Rust subset

As shown in section 4.2, none of the existing semantic models of Rust satisfy our requirements for the proof of semantic preservation in the Ohua compiler. This is why, as part of our proof of concept, we define the syntax and operational semantics for our own subset of Rust, μ_{Rust}.

One of the objectives of the initiative to verify the Ohua compiler is to establish clearly defined subsets of the supported languages. And while μ_{Rust} is too restrictive to serve any practical purpose other than for a proof of concept it is a first step towards specifying Ohua's Rust subset. Originally our subset was closely aligned with RustBelt's λ_{Rust}, which we detail in section 4.2. This would have allowed us to use λ_{Rust}'s type system and, in future work, allowed for easy integration with RustBelt's operational semantics. However, we ended up modeling our subset directly after the Rust AST. We made this decision to be aligned more closely with the Ohua compiler, which operates on the Rust AST directly since as a source-to-source compiler it needs to generate Rust code again. Defining our own subset instead of using an established one means that we also have to define our own type system, which we detail in chapter 5.

Many features of the Rust AST, such as `SpanData` or `TokenStream`, are implementation specific, we excluded those completely. For our transformation we restrict ourselves to features that allow the creation of basic state threads. We tried to keep μ_{Rust} as simple as possible while still largely aligning with the Rust AST. Naturally trade-offs arose between simplicity and strictly adhering to the Rust AST. For many of those trade-offs we ultimately chose the simple version. For example, in the Rust AST assignment is an expression whereas in μ_{Rust} it is a statement. Making this change did not significantly reduce the number of programs we could model with μ_{Rust} but greatly improved the simplicity of both our operational semantics and our proof of semantic preservation.

Another reason for choosing simplicity were restrictions imposed by Coq's termination checker. Every function in Coq must be shown to terminate. This implies that Coq is not Turing-complete. The termination checker's rules by which termination is decided are very rigid, a fact that often resulted in us having to adapt our definitions to accommodate for those restrictions. One such example is the extensive use of identifiers in expressions. Originally many of those used to be sub-expressions but because the termination checker

did not accept the recursive structure of some functions operating on these expressions we chose to use identifiers instead. For trade-offs where choosing the simplified solution could have restricted extensibility for future work we chose the extensible version. This applies for example to `Lit`, which each only have one data constructor but could be extended in future versions of this proof.

4.3.1 Syntax of μ_{Rust}

The grammar of μ_{Rust} is given in figure 4.1. It makes use of the Coq standard library types `string`, `list` and `int`, as implemented in `BinInt`. At its core, μ_{Rust} is an expression language on integers. One minor difference to the original Rust abstract syntax tree (AST) is the continuation style of statements. This aligns closely with Ohua's λ-calculus-based IR. A μ_{Rust} program is a tree with a `Stmt` as its root node. Only one primitive datatype, `Int`, is included for simplicity and because more are not needed for our argument. For the same reason, several binary operations are left out. Support for modules and libraries is not included. Furthermore, const, macros, generics, impls, traits, tuples, enums and unions are not part of this work. While structs are part of the type system, they can't be created with a struct expression. Instead, as Ohua's programming model commands, they can be created as return values from functions and are then operated on by methods. Functions and structs exist only on the type level and not as expressions. Because of the syntactic nature of method and function call semantics, which we describe in section 4.4, struct and function expressions are not needed.

Identifiers are simplified as well to exclude paths and namespaces. All patterns in μ_{Rust} are identifiers, which means that matches are not possible. Variable assignment is only supported as a standalone statement and not as a static assignment or (sub-)expression. Because control flow structures such as conditionals and loops are excluded, our subset is not Turing-complete. Note that the lack of these structures means that useful applications of STCLang's `smap` cannot arise from μ_{Rust}. Our subset includes two kinds of functions: pure ones and methods operating on structs.

Each μ_{Rust} program has two parts: the environment, given as a map of identifiers to types and the algorithm, the actual program. This reflects Ohua's separation of libraries and algorithms. For our transformation only a function's type signature is needed. μ_{Rust} does not support closures so any mutable state has to be passed as a struct argument to a method call.

4.4 Operational semantics

Because μ_{Rust} is an expression language designed to be evaluated in Coq, we define our semantic model directly by implementing an evaluation function, as described in section

BinOp	::=	`Add` | `Sub`	
Type	::=	`Int`	
	|	`Fn` (*list string*) (*list Type*) *Type*	function with parameter identifiers, parameter types and return type
	|	`Struct` (*list string*) (*list Type*)	struct with member identifiers and types
	|	`Unit`	for functions with no return value
	|	`Err`	
Lit	::=	`Int` *int*	
Expr	::=	`Lit` *Lit*	
	|	`BinOp` *BinOp string string*	
	|	`Path` *string*	variable
	|	`Call` *string* (*list Expr*)	function call with arguments
	|	`MethodCall` *string Expr* (*list Expr*)	method call with struct identifier, member identifier and arguments
	|	`Err`	
Mutability	::=	`Mut` | `NotMut`	
Stmt	::=	`Expr` *Expr*	expression
	|	`Let` *string Mutability Type Expr Stmt*	variable binding with continuation statement

Fig. 4.1: Grammar of μ_{Rust}

2.2.1. This evaluation function has as arguments the program to be evaluated, given by a single `Stmt`, as well as the environments for mutability, type, variable definitions. The data structure implementing an environment is a map that is ordered by the order of entry. For an in-depth look at how the environment is implemented, see section 2.2.1. Our evaluation makes use of mutability and type analysis, which we introduce before defining the operational semantics.

4.4.1 Mutability analysis

Mutability inference is the process of determining whether a value may be mutated or not. It is essentially a helper function for type checking and evaluation. In Rust, mutability information is part of the type system, so mutability analysis is done entirely on the type level. In Rust, each variable declaration has mutability information associated. In μ_{Rust} this has to be annotated explicitly. In Rust, if no annotation is given, the value is treated as immutable by default. In μ_{Rust} only values that can be operated on by state threads may be mutable. There exists only one mutable type in our implementation: `Struct`. In this thesis, we use the term *state* to refer to `struct` values. Structs can only be created by function or method calls. This means that the mutability analysis only has to check function and method calls as well as variables, which could hold the result of a call.

To determine mutability, we simply have to do a lookup in our mutability environment. This is done on function calls, to ensure that no argument is mutable, on method calls, to ensure that no argument other than the state is mutable, and on `Let` statements, to check whether the declared mutability of the new variable and matches the mutability of the assigned value.

A previous version of our implementation had Pointers as part of the type system. This resulted in another use case of mutability analysis: ensuring the correct construction of mutable and immutable pointers. Pointers were ultimately excluded from μ_{Rust}, because Ohua does not have support for pointers. If arrays are implemented in a future version if μ_{Rust}, they too may be mutable. This is because they can be operated on by `smap` state threads.

4.4.2 Type analysis

Type analysis is the process of assigning types to expressions. In our implementation it is used on function and method calls, to determine if the types of the arguments match the types of the function parameters, on binary operations, to check if the operands are integers, and on let statements to check if the declared type annotation matches the type of the value.

Type analysis can be formalized as a set of rules, as shown in section 2.2.2. Figure 4.2 gives the typing rules for μ_{Rust}. Here E stands for the global type environment, while Γ stands for the local one. Functions cannot be defined in Ohua's algorithms, which is why their types can only be located in the global environment. Variables on the other hand can be in located in the local or global environments, as they can be defined in Ohua's algorithms. In the current implementation, variables can only have the type *integer*, *struct* or *error*. Each type check evaluates to one of these types. Wrongly typed expressions all have the error type. These are not shown in our rule system, only correctly types ones are.

The first rule, *Lit*, is straightforward: there exist only integer literals, which are of the type `Int`. The underscore in the type rule stands for any integer value. The rule for `Err` expressions is similarly simple: any error expression is of the error type. Binary operations on integers are themselves of the type `Int`, since our implementation so far only supports operations on integers. Here the underscore can stand for either of the `Add` and `Sub` operations. A `Path` expression has the same type as the variable it looks up. If the variable is not present in the environment, the `Path` expression has the type `Err`. If the argument and parameter types match, then the type of any `Call` expression is the return type of the called function. The same goes for `MethodCall` expressions, with the state being the first of the function arguments.

Note that through calling the method with the state as a bare argument, we move the full ownership of the state into the function. If we want to use the state after the method call, we have to return it from the method. This corresponds to Ohua's restriction, that state may only be passed to and from methods as an argument and return value. Note that any method not returning the state "consumes" it. The state can then not be used after the method call. This is different from other values, e.g. integers, which are copied when passed as arguments.

From the typing rules given in Figure it is almost trivial to derive a Coq implementation. However, there is one caveat. When the type check a function call, we check if the argument and parameter types match. To do so, we implement a function, `ty_eq`, which checks the equality of two types. This is not trivial. $Type$ is a *co-inductive* definition, because the data constructor of `Fn` contains the field ($list\ Type$). This means that ($list\ Type$) and $Type$ are inductively defined in terms of each other. Not only that, because the data constructor of `Fn` also has a $Type$ field, $Type$ is inductively defined on itself. The same goes for ($list\ Type$). If the list is not empty, it is defined as a tuple of a $Type$ element, the *head*, and a *list* of $Types$, the *tail*. Therefore, ($list\ Type$) is also inductively defined in terms of $Type$, in the *head*, and itself, in the *tail*. These relationships are visualized in Figure 4.3.

When we define a function `ty_list_eq`, it recursively calls itself and `ty_eq`. `ty_eq` again recursively calls itself and `ty_list_eq`. Coq's termination checker can't handle this

$$\vdash \texttt{Lit}\ (\texttt{Int}\ \_) : \texttt{Int} \qquad (Lit)$$

$$\frac{E,\ \Gamma \vdash x : \texttt{Int} \qquad E,\ \Gamma \vdash y : \texttt{Int}}{E,\ \Gamma \vdash \texttt{BinOp}\ \_\ x\ y : \texttt{Int}} \qquad (BinOp)$$

$$\frac{E,\ \Gamma \vdash x : \tau}{E,\ \Gamma \vdash \texttt{Path}\ x : \tau} \qquad (Path)$$

$$\frac{E \vdash f : \texttt{Fn}\ [\tau_1; \cdots ; \tau_\text{n}]\ \tau_\text{ret} \qquad E,\ \Gamma \vdash x_1 : \tau_1 \quad \cdots \quad E,\ \Gamma \vdash x_\text{n} : \tau_\text{n}}{E,\ \Gamma \vdash \texttt{Call}\ f\ [x_1; \cdots ; x_\text{n}] : \tau_\text{ret}} \qquad (Call)$$

$$\frac{E \vdash \tau_s :: f : \texttt{Fn}\ [\tau_s; \tau_2; \cdots ; \tau_\text{n}]\ \tau_\text{ret} \qquad E,\ \Gamma \vdash s : \tau_s \qquad E,\ \Gamma \vdash x_2 : \tau_2 \quad \cdots \quad E,\ \Gamma \vdash x_\text{n} : \tau_\text{n}}{E,\ \Gamma \vdash \texttt{MethodCall}\ s\ f\ [x_2; \cdots ; x_\text{n}] : \tau_\text{ret}} \qquad (MethodCall)$$

$$\vdash \texttt{Err} : \texttt{Err} \qquad (Err)$$

Fig. 4.2: Typing rules of μ_Rust

$$\begin{array}{c} \mathit{Type}\ \circlearrowleft \\ \downarrow\uparrow \\ (\mathit{list}\ \mathit{Type})\ \circlearrowleft \end{array}$$

Fig. 4.3: Inductive definitions with `Fn`

construct, because a cyclic implication, or equivalence, exists. If `ty_list_eq` terminates, then `ty_eq` terminates and if `ty_eq` terminates, then `ty_list_eq` terminates.

To deal with this paradox, we introduce a helper argument, commonly called `fuel` [VLP17]. `fuel` is of type `nat`, a natural number, and is given as a parameter to `ty_eq` and `ty_list_eq`. Each invocation of `ty_eq` or `ty_list_eq` decreases it by one. If it reaches zero, we run out of fuel and `ty_eq` or `ty_list_eq` return `false`. This causes our type check to return an error type. Because this argument is always decreasing, it satisfies the termination checker.

This workaround, which our implementation also uses for the recursive `Stmt` and `Expr` evaluation, has one drawback: we have to have a fuel argument in all functions which call functions with a fuel argument. This means that we have to carry `fuel` through all the implementation. Our proof too has to be adapted to `fuel`. We have to parameterize it with the quantifier `forall` (∀) `fuel`. To make our proof work, we have to ensure that `ty_eq` and `ty_list_eq` are called exactly as many times in the operational semantics of μ_Rust as they are called in the transformation and the operational semantics of the λ-IR. Otherwise there exists (∃) a value for `fuel`, where one of the semantics of μ_Rust and the λ-IR returns an error value while the other does not.

4.4.3 Evaluation

Now we have all the tools to define the evaluation functions for *Expr* and *Stmt*. Both return a maximally evaluated expression. Evaluating a maximally evaluated expression again yields the same expression. We look at the evaluation function for expressions first.

Evaluating `Lit` is trivial. It is already maximally evaluated, so we don't have to perform any transformations on it. The same applies to `Err`. The evaluation function for `Path` looks up the variable in the value environment and returns the associated value. If no value is found it returns `Err`.

The semantics function of `BinOp` is simple. First it performs a type check on the operands. If both are of the type `Int`, it computes the binary operation and returns a `Lit` with the computed integer value. Else it returns `Err`. Here we see an instance of how our operational semantics differs from Rust's actual ones. We use Coq's built-in arithmetic, which uses a different integer type than Rust. However, such details are not critical to our proof and do not significantly weaken our results.

`Call` expressions are where the type and mutability checkers comes into action. If the parameter types do not match the argument types or if at least one argument is mutable, `Err` is returned. If the arguments are well-typed and immutable, they are evaluated one by one. The function body is not evaluated. `Call` expressions operate on pure functions. Pure functions are referentially transparent. This means that it makes no difference to the outcome of the computation if they are evaluated or not. We chose not to implement function evaluation, because it would not have contributed to the argument of semantic preservation and have made the proof more complex.

The type check in function signatures serves another purpose. Structs are heap allocated. Every time they are passed as mutable state to a method or as an immutable argument to a function or method, their ownership is moved into the function or method. For out implementation of syntactic function and method calls, this means that they are invalidated in the global scope. This invalidation can be achieved by setting their their type to `Err`. The ownership of the struct can be returned to the global scope by returning the (modified) struct. If the called function of method returns no value, the struct is "consumed". Note that no information has to be lost through this restriction, because even for multiple argument structs, which are "consumed", one could build a single struct of a type which aggregates all the consumed input information and return it. This state consumption is modeled entirely on the type system and done at compile time, which in our case corresponds to the μ_{Rust} evaluation and the transformation to the λ-IR. In Rust this is also done at compile time. After all, mutability information is part of the type system.

`MethodCall` expressions are treated similarly as `Call` expressions. Its arguments are type and mutability checked and evaluated. The state is also type and mutability checked. If the type does not match the parameter type, `Err` is returned. If the state is immutable, the `MethodCall` is transformed into a `Call`. The method body is not evaluated. This behavior makes sense not just for pure functions, but also for our restricted stateful ones. The ownership type system, of which we implement moves by state invalidation, gives a strong enough semantics for our proof. This is because state invalidation simulates that a struct is mutated method by reassigning it as the return value. For this to hold, we impose the additional restriction that mutable structs must have globally unique names. This ensures that the order of moves and, therefore, of possible mutations is made explicit. We make use of this property in our proof when showing that the order and arguments of stateful method calls are unaffected by the transformation.

There are two kinds of statements. `Expr` simply wraps around an expression and acts like a return statement because it has no continuation attached. The value of an `Expr` statement is the value of its expression.

`Let` statements are used to introduce new variables and to chain multiple statements. When a new variable is created, the annotated type and mutability is checked against the type and mutability of the assigned expression. If the type does not match or an immutable expression is assigned to a variable declared as mutable, the statement evaluates to `Err`. Reassignment of variables can be simulated by creating a new binding with the same name. Just as with function calls, if `Let` is used to assign a struct, the ownership is moved to the assignee and the original owner is invalidated. μ_{Rust}'s statements are similar to Rust's blocks. The fact that each statement must be terminated with an expression, which is the return value, supports this notion.

4.5 Evaluating an example program

Listing 4.6 shows a valid Rust program that uses only primitives included in our Rust subset. This program contains four functions: two pure ones, `add` and `Counter::new`, and two that operate on state, `Counter::increment` and `Counter::get_value`. This example has been adapted from Listing 3.1.

We can transform this program by hand into the μ_{Rust} AST representation, which is shown in Listing 4.7. Note that the program has been split into two parts: the environment and the algorithm. The algorithm contains the function body of `main` represented by statements linked by continuations. The environment holds a map from identifiers to types. Note that the order in which the types have been added to the environment reflects the environment's stack structure. The value and mutability environments are empty at the beginning of the algorithm.

The evaluated program is given in Listing 4.8.

```
fn add(a: i32, b: i32) -> i32 { a + b }
struct Counter { c: i32 }
impl Counter {
  fn new() -> Counter { Counter {c: 0} }
  fn increment(mut self, i: i32) -> Counter { self.c += i; self }
  fn get_value(self) -> i32 { self.c }
}
fn main() {
  let n: i32 = 2;
  let m: i32 = 3;
  let mut c1: Counter = Counter::new();
  let x: i32 = add(1, n+m);
  let mut c1: Counter = c1.increment(x);
  c1.get_value();
}
```

Listing 4.6: μ_{Rust} example program

Type environment:

```
(add_item
  "Counter::get_value"
  (FnTy [(StructTy ["c"] [IntTy])] IntTy)
(add_item
  "Counter::increment"
  (FnTy [(StructTy ["c"] [IntTy]), IntTy] (StructTy ["c"] [IntTy]))
(add_item
  "Counter::new"
  (FnTy [] (StructTy ["c"] [IntTy]))
(add_item
  "add"
  (FnTy [IntTy, IntTy] IntTy)
(empty_map Ty)))))
```

Mutability environment:

```
(add_item "Counter::get_value" NotMut
(add_item "Counter::increment" Mut
(add_item "Counter::new" Mut
(add_item "add" NotMut
(empty_map Mutability)))))))
```

Algorithm:

```
(Let "a" false IntTy
  (LitE (Int 2))
(Let "b" false IntTy
  (LitE (Int 3))
(Let "c1" true (StructTy ["c"] [IntTy])
  (CallE "Counter::new" [])
(Let "x" false IntTy
  (CallE "add" [(LitE (Int 1)), (BinOpE AddOp "a" "b")])
(Let "c1" true (StructTy ["c"] [IntTy])
  (MethodCallE "Counter::increment" (PathE "c1") [(PathE "x")])
(ExprS
  (MethodCallE "Counter::get_value" (PathE "c1") [])))))))
```

Listing 4.7: μ_{Rust} example program AST

Type environment:

```
(add_item "c1" Err
(add_item "c1" (StructTy ["c"] [IntTy])
(add_item "c1" Err
(add_item "x" Int
(add_item "c1" (StructTy ["c"] [IntTy])
(add_item
  "Counter::get_value"
  (FnTy [(StructTy ["c"] [IntTy])] IntTy)
(add_item
  "Counter::increment"
  (FnTy [(StructTy ["c"] [IntTy]), IntTy] (StructTy ["c"] [IntTy]))
(add_item
  "Counter::new"
  (FnTy [] (StructTy ["c"] [IntTy]))
(add_item
  "add"
  (FnTy [IntTy, IntTy] IntTy)
(empty_map Ty)))))
```

Mutability environment:

```
(add_item "x" NotMut
(add_item "c1" Mut
(add_item "Counter::get_value" NotMut
(add_item "Counter::increment" Mut
(add_item "Counter::new" Mut
(add_item "add" NotMut
(empty_map Mutability)))))))
```

Value environment:

```
(add_item "c1"
  (MethodCall "Counter::increment" (Call "Counter::new" [])
    [(CallE "add" [(Lit (Int 1)), (Lit (Int 5))]))
(add_item "x" (CallE "add" [(Lit (Int 1)), (Lit (Int 5))])
(add_item "c1" (Call "Counter::new" [])
(empty_map Expr))))))
```

Algorithm:

```
(RD.MethodCallE "Counter::get_value"
  ((MethodCall "Counter::increment" (Call "Counter::new" [])
    [(CallE "add" [(Lit (Int 1)), (Lit (Int 5))])])))
```

Listing 4.8: μ_{Rust} example program AST

Transformation and Proof

5

In this chapter we cover the transformation from μ_{Rust} to the λ-IR and the proof of semantic preservation. The proof's outline is given in Figure 5.1. We show that for every μ_{Rust} program the semantics are equal to the semantics of the transformed program. The semantics for the λ-IR have already been defined in section 3.3. Similarly, the semantics of μ_{Rust} have been defined in section 4.4.

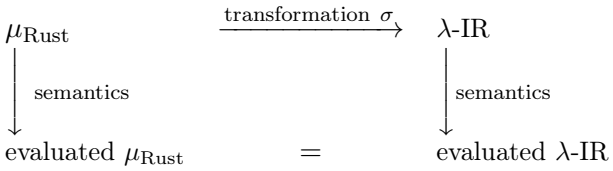


Fig. 5.1: Proof outline

5.1 λ-IR code from μ_{Rust}

Because the features of the λ-IR and μ_{Rust} largely overlap, deciding which parts of the λ-IR should be mapped to which parts of μ_{Rust} is straightforward. However, this mapping is only one of two functions the transformation serves. The other function is to determine, which programs are correct in Ohua's programming model and which ones are not.

As mentioned in section 3.1, Ohua is currently being rewritten. This is partly motivated by the fact, that Ohua's error handling has so far been done in all stages of the compiler. The rewrite aims to alter this so that all errors in the program being transformed are detected in the first stage if the compiler. This is useful for extensibility, because we have a single source of truth of which Rust program are supported.

For this it is crucial to have a clear notion of which programs can be transformed and which can not. One of the motivations for this thesis is to determine a small subset of Rust, for which it is certain that it is supported. This information can be used to implement the front-end of the compiler. We define the subset supported in our implementation in two different ways. First, we give purely syntactic constraints, simply by choosing the rules for our grammar. Second, out semantics, in particular the type checker rejects ill-formed programs, evaluating them to an error expression.

Our transformation is a mapping from μ_{Rust} to the λ-IR. Its informal rules are given in Listing 5.2. Data constructors of μ_{Rust} are annotated with a R, data constructors of

μ_{Rust}	$\xrightarrow{\text{transformation } \sigma}$	λ-IR
$\texttt{Err}_{\text{R}}$	$\longmapsto$	$\texttt{Lit}_\lambda\ \texttt{Err}_{\text{R}}$
$\texttt{Lit}_{\text{R}}\ i$	$\longmapsto$	$\texttt{Lit}_\lambda\ (\texttt{Lit}_{\text{R}}\ i)$
$\texttt{BinOp}_{\text{R}}\ op\ e_1\ e_2$	$\longmapsto$	$\texttt{Lit}_\lambda\ (\texttt{BinOp}_{\text{R}}\ op\ e_1\ e_2)$
$\texttt{Path}_{\text{R}}\ s$	$\longmapsto$	$\texttt{Var}_\lambda\ s$
$\texttt{Call}_{\text{R}}\ f\ [e_1;\ \ldots;\ e_n]$	$\longmapsto$	$\texttt{FunApp}_\lambda\ (\texttt{PureFun}_\lambda\ f\ [p_1;\ \ldots;\ p_n])$ $[\sigma(e_1);\ \ldots;\ \sigma(e_n)]$
$\texttt{MethodCall}_{\text{R}}\ f\ e_{\text{st}}\ [e_1;\ \ldots;\ e_n]$	$\overset{e_{\text{st}}\text{ immutable}}{\longmapsto}$	$\texttt{FunApp}_\lambda\ (\texttt{PureFun}_\lambda\ f\ [p_s; p_1;\ \ldots;\ p_n])$ $[\sigma(e_{\text{st}}); \sigma(e_1);\ \ldots;\ \sigma(e_n)]$
$\texttt{MethodCall}_{\text{R}}\ f\ e_{\text{st}}\ [e_1;\ \ldots;\ e_n]$	$\overset{e_{\text{st}}\text{ mutable}}{\longmapsto}$	$\texttt{FunApp}_\lambda\ (\texttt{STFun}_\lambda\ f\ p_s\ [p_1;\ \ldots;\ p_n])$ $[\sigma(e_{\text{st}}); \sigma(e_1);\ \ldots;\ \sigma(e_n)]$
$\texttt{Expr}_{\text{R}}\ e$	$\longmapsto$	$\sigma(e)$
$\texttt{Let}_{\text{R}}\ s\ m\ t\ e_{\text{val}}\ s_{\text{cont}}$	$\longmapsto$	$\texttt{Let}_\lambda\ s\ \sigma(e_{\text{val}})\ \sigma(s_{\text{cont}})$

Fig. 5.2: Mapping from μ_{Rust} to the λ-IR

the λ-IR with a λ. The transformation re-uses many components of the rust semantics implementation, e.g. type and mutability checking. These components are covered in section 4.4. Our transformation is parameterized with environments for mutability and for type information. We do not evaluate any expressions, which is why we do not need a value environment. This is the complement to the evaluation function of the λ-IR. The λ-IR in our implementation does not have any type or mutability information embedded, as those are checked during the transformation. Therefore, the evaluation function for the λ-IR only has the value environment as a parameter.

The transformation of `Err`, `Lit` and `BinOp` is a simple embedding. We embed `BinOp` as a whole Rust expression, because the λ-IR does not contain any data constructor for it. This is a simple solution, but has the drawback that our operand sub-expressions are not transformed. This is because the expression is evaluated as a whole by the μ_{Rust} evaluation function.

An alternative to this embedding would have been to define a pure function for each binary operation. In this representation we could transform the operand expressions. However, the question arises of how the transformed function is evaluated. Simply defining a function `"+"` for example won't suffice because in our implementation function bodies are neither implemented nor evaluated. We could match against literals with embedded binary operations in the λ-IR evaluation, but that would go against the principle of excluding a `BinOp` data constructor in the λ-IR. We excluded `BinOp` in our formalized λ-IR, because it is not present in Ohua.

Transforming `Path` expressions is a trivial case of switching data constructors. Transforming `Call` is more involved. The representation for functions in the λ-IR differs from the one in μ_{Rust} in one way: the parameter identifiers are explicitly annotated. In μ_{Rust} they are part of the function type, stores in the type environment. To transform a `Call` expression,

```
(LetL "a" (LitL (LitR (Int 2)))
(LetL "b" (LitL (LitR (Int 3)))
(LetL "c1" (FunAppL
  (PureFun "Counter::new" [])
    [])
(LetL "x" (FunAppL
  (PureFun "add" ["a", "b"])
    [LitL (LitR (Int 1)), LitL (BinOp AddOp "n" "m")])
(LetL "c1" (FunAppL
  (STFun "Counter::increment" "s" ["i"])
    [VarL "c1", VarL "x", nil]
(FunAppL
  (STFun "Counter::get_value" "s" [])
    [VarL "c1", nil])))))))
```

Listing 5.1: Transformed example program

we look up its parameter identifiers in the type environment and construct a function application of a pure function. We also transform the arguments. The transformation of a `MethodCall` is similar to the one of `Call` but has one extra step: checking if the state arguments is mutable. If it is, the method call is transformed to a stateful function call, else to a pure one. If the transformation fails, it returns a `Lit Err` value to indicate the error.

To transform an `Expr` statement, we just have to transform the embedded expression. For `Let` statements we first transform the value expression. Then we check its type and mutability against the annotated ones. We add them to the type and mutability environments. Finally, we transform the continuation statement with the updated environments.

As an example for a transformation, consider the μ_{Rust} program in Figure 4.7. Transforming this program to the λ-IR yields the one in Figure 5.1.

5.2 Proof

In Coq, we prove invariants about our programs, and in this thesis the semantics of μ_{Rust} and the λ-IR is the invariant prove. Our argument of semantic preservation can only be as complete as the semantic models. For example, function calls are not evaluated beyond the type and borrow system in our implementation. This relies on the assumption, that only the state arguments of method calls, whose ownership is moved into the method, are mutated. This is ensured by Rust's type system. If we wanted our proof to hold without this assumption, we would have to implement function evaluation.

```
Theorem transform_stmt_correct :
forall (s : Stmt) (fuel : nat)
(val_env : Map Expr) (type_env : Map Ty) (mut_env : Map Mutability),
eval_expr val_env (transform_stmt fuel type_env mut_env s) = eval_stmt fuel val_env type_e
```

Listing 5.2: Theorem of semantic preservation in Coq

Note that for transformations over multiple internal representations, only the semantics for the first and last representation have to be implemented. Ohua is a source to source compiler, so a complete proof would have to implement just one operational semantics, Rust's. The transformation parallelizes the input program using the Ohua runtime. For this it makes use of concurrency constructs from Rust's standard library. Our semantics would have to implement some definition of execution of these concurrent constructs.

One way to do this is to look at the data flow graph. Recall that independent nodes can be executed in parallel. The data flow graph for Ohua is a directed acyclic graph (DAG). This is because all loops, which would introduce back-edges, are resolved to an `smap` application that is represented as a single node. Szpilrajn's extension theorem states that every DAG can be linearized [Szp30]. This means that we could look at all possible linearizations of the transformed program and prove semantic equivalence for each of them.

A simpler approach would use the theorem that Rust's type system disallows any invalid use of the standard libraries' concurrency modules [Jun+19]. This result means that it may be enough to show that there exists one linearization, for which the semantics are equivalent to the semantics of the input program. For our formal proof we don't have to consider concurrent execution yet, because at the stage of the λ-IR, Ohua has not introduced any parallelism.

The result we are trying to prove is that evaluating a `Stmt` with μ_{Rust}'s evaluation function yields the same result as transforming it and the evaluating it with the λ-IR's evaluation function. This theorem is shown in its formalized version in Listing 5.2.

Our proof is by structural induction over statements. We also have to perform nested inductions over expressions and the helper variable `fuel`. As explained in section 4.4.2, the co-inductive definitions of our `Type` and `Expr` types are impossible to recursively reason about in Coq without a helper variable. This is why we introduced `fuel`. We found no way to entirely prove the correctness of the transformation for these recursive definitions. We give a proof sketch in the implementation.

We implemented a complete proof for a slightly modified version of μ_{Rust}. In this version, expressions do not directly contain sub-expressions. Instead, they are defined using variable identifiers. Any program of the original μ_{Rust} can be transformed into a program in the modified μ_{Rust} without loss of semantics by introducing a new variable for each sub-expression.

In the simplified version of μ_{Rust}, only the `Let` data type is defined inductively. This means that we have to perform a structural induction over `Let`. The base case is `Let`'s non-inductive `Expr` data constructor. Here we have to prove that the transformation is correct for the embedded expression. In the induction step, we have to prove the correctness for the statement:

```
Let name mutability type expression cont
```

The induction hypothesis is given in Listing 5.3. Note that the continuation statement `cont` is not quantified with `forall`. The continuation statement, for which the induction hypothesis holds, has to be precisely the one in our `Let` statement of the induction step.

```
forall (fuel : nat)
(val_env : Map Expr) (type_env : Map Ty) (mut_env : Map Mutability),
eval_expr val_env (transform_stmt fuel type_env mut_env cont) =
eval_stmt fuel val_env type_env mut_env cont
```

Listing 5.3: Induction hypothesis

We do not give the entire proof in this section, because it is too large. To show how our proof works, consider the example:

```
Let name mut type (Path var) cont
```

For this we have to prove the goal given in Listing 5.4. In this goal, some proof steps have already been performed. For example, we have already proven the mutability check.

```
eval_expr val_env
  (let (type_env', expr') := transform_expr type_env mut_env (Path var) in
    Let name expr'
      (transform_stmt fuel
        (add_item name type type_env')
        (add_item name mut mut_env)
        cont)) =
let (type_env', expr') := eval_expr val_env type_env mut_env (Path var) in
  eval_stmt fuel
    (add_item name expr' val_env)
    (add_item name type type_env')
    (add_item name mut mut_env)
    cont
```

Listing 5.4: Initial goal

It can be simplified into the goal in Listing 5.5 by unfolding the definitions of `transform_expr` and `eval_expr`.

```
eval_expr
 (add_item name
    (match val_env var with
      | Some expr' => expr'
      | None => Err
    end)
    val_env)
  (transform_stmt fuel
    (add_item name type type_env')
    (add_item name mut mut_env)
    cont)) =
let (type_env', expr') :=
  (match val_env var with
  | Some expr' => (type_env, expr')
  | None => (type_env, expr')
  end)
in
  eval_stmt fuel
    (add_item name expr' val_env)
    (add_item name type type_env')
    (add_item name mut mut_env)
    cont
```

Listing 5.5: Goal with unfolded definitions

We do a case analysis on `val_env var`. If there is no entry in the value environment, both sides of the equation yield an error expression. If an entry exists, we get the goal shown in Listing 5.6.

```
eval_expr (add_item name expr' val_env)
  (transform_stmt fuel
    (add_item name type type_env')
    (add_item name mut mut_env)
    cont) =
eval_stmt fuel
  (add_item name expr' val_env)
  (add_item name type type_env')
  (add_item name mut mut_env)
  cont
```

Listing 5.6: Goal after case analysis

We can apply our induction hypothesis to yield the final goal in Listing 5.7, which is obviously true.

```
eval_stmt fuel
  (add_item name expr' val_env)
  (add_item name type type_env')
  (add_item name mut mut_env)
  cont =
eval_stmt fuel
  (add_item name expr' val_env)
  (add_item name type type_env')
  (add_item name mut mut_env)
  cont
```

Listing 5.7: Final goal

```
No more subgoals.
```

Conclusion 6

In this thesis we have demonstrated one approach to formal verification in the Ohua compiler. Our approach involves defining semantics for the host language and for the IR we want to prove semantic preservation for. This is fundamentally constrained by the semantic model of the host language. Because, as we showed in section 4.2, no suitable semantic model for Rust exists, we set out to define our own. For this thesis we spent an estimated 80% of the coding time on defining μ_{Rust}'s syntax and semantics. Part of the reason for that was that Ohua has an imprecise definition of the subset of Rust it works on, given through the implementation. For the definition of μ_{Rust} we had to find a subset that is both large enough to serve for a reasonable proof of concept but is small enough to fit the scope of this thesis. Defining operational semantics for Rust's semantic model is more complicated than for most languages, because of the unique ownership type system and the borrowing semantics. These features make Rust a perfect suitor for Ohua but complicate the implementation of the operational semantics.

We found theorem proving in Coq to be effective for reasoning about programs. Thanks to the interactive IDE tooling, it is surprisingly intuitive. Gallina is simple to learn if you have experience in a statically typed pure functional programming language, like Haskell or ML. The Ohua compiler is implemented in Haskell, which resembles Coq's Gallina. This meant that we could translate Ohua's source code easily when needed. The discrete nature of compilers makes them well suited for the use with theorem provers, such as Coq. The biggest challenge in working with Coq is satisfying the termination check. This is especially complicated for co-inductive data types. In multiple instances we had to adapt our implementation and even specification to account for Coq's behavior. However, in no such case had the changes much impact on the project, small "hacks" sufficed.

6.1 Future work

We have concluded that proving Ohua's semantics by defining a semantic model of Rust is too much work to be a single project. That does not mean that all is lost, as there are approaches to verifying semantic properties of Ohua, which we cover in this section.

6.1.1 Extend features of μ_{Rust}

One way to build on the work of this thesis is to gradually add features to μ_{Rust} and thus to increase the scope of this thesis' proof. The most critical feature missing is control flow,

i.e. `if` and/or `loop` expressions. In this scenario we would have to extend our λ-IR model with `if` and/or `smap` constructs. Especially introducing `loop` expressions is a prerequisite to making our proof more meaningful as we could extend it to cover `smap`'s parallelization capabilities.

6.1.2 Pre-defined semantic models

As we laid out in section 4.2, there currently exists no sufficient semantic model to prove semantic preservation for all of Ohua. Still, in the future more complete models of Rust might become available which we could use for our proof. For example, the RustBelt project might be extended to operate on High-level IR (HIR), which is close to the AST. Or, if Oxide's implementation continues, we could use it once the operational semantics have been implemented. As Rust's type system is well suited for Ohua, we believe a proof using a pre-defined model of Rust is achievable with reasonable effort. Note that Ohua's algorithms are implemented in only a small subset of Rust, so the semantic model would not have to encompass all of Rust, but only our subset. Traits for example are a complex language feature of Rust, that has not been formalized in any semantic model known to us, but it is not needed for Ohua.

Although we ruled out RustBelt's semantic model as a basis for our proof of semantic preservation of the complete Ohua compiler, we could use it for a limited proof. RustBelt's semantics are defined on the Rust compiler's Mid-level IR (MIR), which represents a control-flow graph. For our proof we could translate the MIR directly to Ohua's dataflow IR, which gives us a tool to prove the correctness of optimizations on the dataflow IR. In order to exploit the full potential of Ohua's parallelization we would have to implement an analysis pass on the MIR which can recognize instances of `smap`.

Contrary to our own approach of using Coq's intrinsic functions for operational semantics, Rust's semantics are specified through a transformation from Rust to a core language. This approach has the advantage that the entire Rust language can be semantically modeled. Our own approach of defining operational semantics is bound to run into Coq's language limitations, for example when defining a semantic model for pointer arithmetic.

6.1.3 Implement more transformations

Another approach of future work would be to implement Ohua's dataflow graph representation in Coq. We could then extend the proof to work on the entire Ohua compiler, including optimization and code generation. This was the initial motivation of this thesis and continues to be a valid research goal.

6.1.4 Proof workflow

Once we have extended our proof to work on all stages of the Ohua compiler, we can work on integrating the certified version of Ohua with the actual implementation. This could be done in two different ways.

First, we could generate Haskell code from our Coq source using Coq's integrated code generator [@21f]. We could then link our generated Haskell against IO libraries, e.g. custom parsers and code generators. This approach has the advantage that our code is written from the beginning with Coq's restrictions, which makes the proof less invasive.

Second we could use `hs-to-coq` [Spe+18] to generate Coq code from our Haskell source. This approach has the advantage that we can prototype more quickly in Haskell than in Coq and only adapt the proof for milestone releases. The proof itself would require more work than in our first approach because `hs-to-coq` requires custom annotations for the translation.

In both cases we are limited to using total (i.e. provably terminating) functions. This restriction is imposed Coq and by extension by `hs-to-coq`.

6.1.5 Other host languages

Instead of Rust, for which defining semantics in complex, we could choose a host language supported by Ohua for which semantics are simple to define. This could work for example for Ohua's Lisp-like language or OCaml. Operational semantics for Lisp would be easier to define than for Rust, since the type system alone is less complex. Operational semantics for a large subset of Ocaml have already been defined [Owe08]. This means that the biggest challenge of our proof, defining semantics is already accomplished.

Unfortunately proofs for Lisp or OCaml would not hold as much value as a proof for Rust. In the current version of Ohua, OCaml and Lisp are already deprecated. This means that our goal of integrating the mechanized proof into the compiler development process cannot be achieved easily.

Still, proving semantic preservation in the Ohua compiler for one of these two languages would suffice as a proof of concept. Proving semantics for a different host language than Rust has another advantage: part of the proof can be reused for other host languages, as Ohua's IRs and optimizations are language independent. If we want to adapt the proof for a different host language, we just need to implement the transformations of the new host language to and from Ohua's IR's and give a semantic model. For our new proof we could reuse existing lemmas that prove propositions on the language-independent IR's and transformations.

Because Ohua is internally language-independent, it may be possible to define language-independent semantics directly for its intermediate representations, without needing any host language. The parts of our proof working with only generic semantics are exactly the same as the parts that could be reused for new host languages. Although with this approach our certification would not cover all of the compiler, it would be a basis for future work, since it can be adapted for any host language.

6.1.6 Other proof approaches

One low-hanging fruit to prove the (partial) correctness of the Ohua compiler might be to mechanize the pen-and-paper proof of semantic preservation for the λ-calculus-like languages [Ert+19a] in a proof assistant such as Coq. While not directly integrated in the compiler, such a proof could increase the confidence in Ohua's optimizations conceptually.

We could also prove the partial correctness of the Ohua compiler by limiting our proof of semantic preservation to the type level. An implementation in Coq of Rust's type system exists within the Oxide project [Wei+20]. Proving type preservation may provide a sufficient guarantee of semantic preservation as Rust's type system enforces correct use of mutable state.

Extending the development of Ohua in Haskell with refinement Types via Liquid Haskell [Vaz+14] is another option for partly certifying the correctness of the Ohua compiler. Although it is not likely to verify the entire compiler via automatic deduction, it does give partial correctness guarantees without much effort. For this reason it is already being integrated in the Ohua development process.

Bibliography

[Ada16] Justus Adam. "Control Flow and Side Effects Support in a Framework for Automatic I/O Batching". BA Thesis. Dresden, Germany: TU Dresden, Oct. 2016 (cit. on p. 11).

[Ada19] Justus Adam. "Ohua-powered, Semi-transparent UDF's in the Noria Database". MA thesis. Dresden, Germany: TU Dresden, Nov. 2019 (cit. on pp. 11–13).

[BC04] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Jan. 2004 (cit. on p. 3).

[BC11] Shekhar Borkar and Andrew A. Chien. "The Future of Microprocessors". In: *Commun. ACM* 54.5 (May 2011), pp. 67–77 (cit. on p. 1).

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013 (cit. on p. 3).

[Ert+19a] Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. *Category-Theoretic Foundations of "STCLang: State Thread Composition as a Foundation for Monadic Dataflow Parallelism"*. 2019. arXiv: `1906.12098 [cs.PL]` (cit. on pp. 11, 46).

[Ert+19b] Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. "STCLang: State Thread Composition as a Foundation for Monadic Dataflow Parallelism". In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: Association for Computing Machinery, 2019, pp. 146–161 (cit. on pp. 11, 13).

[EFF15] Sebastian Ertel, Christof Fetzer, and Pascal Felber. "Ohua: Implicit Dataflow Programming for Concurrent Systems". In: *Proceedings of the Principles and Practices of Programming on The Java Platform*. PPPJ '15. Melbourne, FL, USA: Association for Computing Machinery, 2015, pp. 51–64 (cit. on p. 1).

[Ert+18] Sebastian Ertel, Andrés Goens, Justus Adam, and Jeronimo Castrillon. "Compiling for Concise Code and Efficient I/O". In: *Proceedings of the 27th International Conference on Compiler Construction*. CC 2018. Vienna, Austria: Association for Computing Machinery, 2018, pp. 104–115 (cit. on p. 13).

[Gen35] Gerhard Gentzen. "Untersuchungen über das logische Schließen. I". In: *Mathematische Zeitschrift* 39 (1935), pp. 176–210 (cit. on p. 6).

[Jun20] Ralf Jung. *Understanding and evolving the Rust programming language*. Saarbrücken, Germany, 2020 (cit. on pp. 21, 22).

[Jun+19] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. "Stacked Borrows: An Aliasing Model for Rust". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019) (cit. on pp. 23, 39).

[Jun+17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017) (cit. on p. 24).

[Jun+21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. “Safe Systems Programming in Rust”. In: *Commun. ACM* 64.4 (Mar. 2021), pp. 144–152 (cit. on pp. 22, 23).

[Kan+20] Shuanglong Kan, Zhe Chen, David Sanan, Shang-Wei Lin, and Yang Liu. *An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing*. 2020. arXiv: `1804.07608 [cs.PL]` (cit. on p. 24).

[Lu+08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics”. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 329–339 (cit. on p. 1).

[Owe08] Scott Owens. “A Sound Semantics for OCamllight”. In: *Programming Languages and Systems*. Ed. by Sophia Drossopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–15 (cit. on p. 45).

[Ree15] Eric Reed. “Patina: A Formalization of the Rust Programming Language”. MA thesis. Seattle, WA, USA: University of Washington, Feb. 2015 (cit. on p. 24).

[Spe+18] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. “Total Haskell is Reasonable Coq”. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 14–27 (cit. on pp. 3, 45).

[SL05] Herb Sutter and James Larus. “Software and the Concurrency Revolution: Leveraging the Full Power of Multicore Processors Demands New Tools and New Thinking from the Software Industry.” In: *Queue* 3.7 (Sept. 2005), pp. 54–62 (cit. on p. 1).

[Szp30] Edward Szpilrajn. “Sur l’extension de l’ordre partiel”. In: *Fundamenta Mathematicae* 16 (1930), pp. 386–389 (cit. on p. 39).

[VLP17] Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. “A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq”. In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. Haskell 2017. Oxford, UK: Association for Computing Machinery, 2017, pp. 63–74 (cit. on p. 30).

[Vaz+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. “Refinement Types for Haskell”. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’14. ACM, Sept. 2014, pp. 269–282 (cit. on p. 46).

[Wan+18] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. “KRust: A Formal Executable Semantics of Rust”. In: *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 2018, pp. 44–51 (cit. on p. 24).

[Wei+20] Aaron Weiss, Olek Gierczak, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. *Oxide: The Essence of Rust*. 2020. arXiv: `1903.00982` (cit. on pp. 24, 46).

[Wit20] Felix Wittwer. “Ohua as an STM Alternative for Shared State Applications”. MA thesis. Dresden, Germany: TU Dresden, Aug. 2020 (cit. on pp. 1, 11).

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 283–294 (cit. on p. 1).

Webpages

[@18] *Announcing the Formal Verification Working Group*. 2018. URL: https://internals.rust-lang.org/t/announcing-the-formal-verification-working-group/7240 (visited on Apr. 2, 2021) (cit. on p. 24).

[@21a] *coq-of-ocaml*. 2021. URL: https://clarus.github.io/coq-of-ocaml (visited on Apr. 5, 2021) (cit. on p. 24).

[@21b] *K Semantic Framework*. 2021. URL: https://kframework.org (visited on Apr. 5, 2021) (cit. on p. 24).

[@21c] *Networking - Rust Programming Language*. 2021. URL: https://www.rust-lang.org/what/networking (visited on Apr. 1, 2021) (cit. on p. 13).

[@21d] *Oxide*. 2021. URL: https://github.com/aatxe/oxide (visited on Apr. 5, 2021) (cit. on p. 24).

[@21e] *The Coq Proof Assistant*. Version 8.13.1. 2021. URL: https://coq.inria.fr (visited on Apr. 6, 2021) (cit. on pp. 2, 3).

[@21f] *The Coq Reference Manual*. Version 8.13.1. 2021. URL: https://coq.inria.fr/distrib/current/refman (visited on Apr. 6, 2021) (cit. on pp. 3, 45).

List of Figures

List of Listings