Forside

Eksamensinformation
NDAB12006E - Bachelorprojekt i datalogi, Datalogisk Institut - ID:mjt368 (Kristian Knudsen Olesen)

Besvarelsen afleveres af
Kristian Knudsen Olesen
mjt368@alumni.ku.dk

Eksamensadministratorer
DIKU Eksamen
uddannelse@diku.dk

Bedømmere
Martin Elsman
Eksaminator
mael@di.ku.dk
+4535335683

Jesper Andreas Bengtson
Censor
jebe@itu.dk

Besvarelsesinformationer

Titel, engelsk: Etracting Certified Futhark Code From Coq
Tro og love-erklæring: Ja
Må besvarelsen gøres til genstand for udlån: Ja
Må besvarelsen bruges til undervisning: Ja

KRISTIAN KNUDSEN OLESEN
November 1, 2021

EXTRACTING CERTIFIED FUTHARK CODE FROM COQ

SUPERVISOR: MARTIN ELSMAN

BACHELOR PROJECT IN COMPUTER SCIENCE
DEPARTMENT OF COMPUTER SCIENCE,
UNIVERSITY OF COPENHAGEN

Abstract

As the rapid increase in sequential CPU speed seem to have come to and end, there is an ever growing need for parallel computing, in order to meet the still increasing performance demand. The Futhark language is precisely intended to help produce code with a high degree of parallelism, which can be difficult to do manually. In this project, we take a look at how feasible it is to use the MetaCoq and ConCert frameworks to extract certified Futhark code from the proof assistant Coq, for example in order to statically check array indexing so that array bound check can be skipped and performance increased. We will carry out this investigation by using the FuCert framework to implement and extract the maximum segment sum function, as well as prove functional correctness of the implementation. During this process we will be extending the FuCert framework with suited infrastructure for future use.

Contents

SECTION 1

Introduction

Correctness is relevant in all facets of programming; nobody has any use for a program that never does what you expect it to do. For this reason, there must always be a belief in that the program in question does what is expected. However, how firm this belief must be and how important it is that the program does what is expected varies greatly. For a lonely programmer at home that has written their own syntax highlighter for Emacs, the stakes are very low, and as failure comes with no noteworthy cost for him, he can base this belief solely on the fact that he wrote the code himself and thinks that he did a good job. On the other hand, if the software in question controls critical components of a satellite or a plane, then failure can come with a tremendous cost or horrible consequences, and belief in the correctness of the software must be based on more than just the fact that the developers who wrote the code thinks that they are awesome.

There are many methods for verifying that software does what it is supposed to do and often a combination of many methods is used. The most widely used methods are various forms of testing, such as unit testing, integration testing and manual testing. Code review are also used as a method for increasing this certainty, in the belief that if two developers have looked at the code and believe that it is correct, the assurance is greater than if it was just one developer. Now, these methods comes in varying degrees of formality. Code review are a very informal verification, whereas unit testing is more formal. In the end of the spectrum is the concept of formal verification, which is the degree of mathematically proving the correctness of (some aspect of) a program. This proof might be done using software, such as proof assistants, instead of using pen and paper.

In this project, we are going to be concerned with formal verification of Futhark programs, or rather writing certified Futhark programs instead of verifying existing programs. Futhark is a small purely functional language, developed at the Department of Computer Science at University of Copenhagen. The purpose of the language is to generate highly efficient parallel code, in particular, for GPU execution (Graphics Processing Unit). Since CPU manufacturers have reached a physical limit for (sequential) CPU speed, due in particular to heat problems, the need for parallel code is becoming greater and greater. However, writing programs which takes advantage of the possibility for parallel computing is difficult, and doing it effectively even more so. This is one of the main reasons why languages like Futhark are relevant, as they make it easy for the programmer to write programs that leverage a high degree of parallelism.

The reasons for using formal verification in connection with Futhark is that, through this, we might be able to increase performance by removing the need for things like array bounds check, or to verify that certain Futhark requirements are satisfied, such as the monoid criteria. This project will be concerned with verification using the proof assistant Coq. More specifically, we will be looking at whether it is viable to use the certified erasure process from the MetaCoq project (see [11]), and its extension with typing information from the ConCert project (see [3]), to generate certified Futhark

code. This will be done in the context of the FuCert framework by D. Annenkov, which contains a pretty printer from $\lambda^T_\square$to Futhark.

1.1 A use-case scenario

Before we go on, let us, by example, show how how all the things mentioned so far fit together, that is, how a programmer would go about creating certified Futhark code.

The small example we have in mind is the example of vector addition. Say that the programmer would like to have a Futhark implementation of a function on $\mathbb{Z}^2$ that does a counterclockwise 90 degrees rotation, which is certified to return something non-zero if the input was non-zero. Then the programmer would implement this function in Coq,

```
Definition f (c : Z * Z) : Z * Z := let '(x, y) := c in (- y, x).
```

The programmer could then prove the following result in Coq:

```
Lemma f_prop: forall (x y : Z), (x, y) <> (0, 0) -> f (x, y) <> (0, 0).
```

The programmer can then use the FuCert framework to extract the function `f` to Futhark code, which might result in something like

```
let opp (x : i64) = -x

let f (c : (i64,i64)) = match c
case (x, y) ->  ((opp y), x)

let main = f
```

The programmer is then guaranteed that, using this function, the result will be non-zero of the input is non-zero.

1.2 About the project

In this section we provide some relevant information about the project. First, let us mention that all implementation done during the project has been done in Coq, in the FuCert framework. The code can be found in the following fork of the official FuCert Git repository[1]

`https://github.com/okknudsen/futhark-extract`

on the `ba` branch. More information about the implementation can be found in Section A.

Before we go on, let us mention that, during the project, we have implemented two versions of the infrastructure (described in Section 4) and the maximum segment sum function (described in Section 5). One of these uses the array model described in Section 4.3 and the other uses plain lists. In this report, we will only focus on the former of the two, as we feel that this version is the one that is most interesting and is most easy to use.

[1] `https://github.com/annenkov/futhark-extract`

1.2.1 Purpose and aim

The purpose of this project is to test whether it is feasible to use the Coq proof assistant for generating certified Futhark code using the MetaCoq and ConCert frameworks. More specifically, to use the FuCert[2] framework, which employs the erasure procedures from MetaCoq and ConCert, to implement, certify and extract Futhark functions. This test will be done by going through the process for an example program that finds the maximum segment sum of an array, extending the FuCert framework along the way to accommodate this example implementation.

The implementation of the maximum segment sum function should be a good semblance of how a user would create certified Futhark using FuCert, and should shed some light on what the obstacles, advantages and disadvantage are of doing so through FuCert, which uses ConCert for extraction.

1.2.2 Reading guide

Let us give an overview on how the project is structured. In Section 2 we give a brief introduction to Futhark and begin discussing aspects of it that are relevant to this project. This is mainly consists in introducing some of the array combinators, describing the monoid criteria some of them have, and then give examples of what properties that could be relevant to verify in a Futhark program.

In Section 3 discuss some aspects of the proof assistant Coq that can cause problems, or at least complications, in the use we put it to. These have a lot to do with equality and how Coq handles equality for subset types and functions.

In Section 4 we will introduce the FuCert framework, explaining how it relates to MetaCoq and ConCert, and discuss how we extended it during this project. A large part of this has to do with how one should model the Futhark arrays, and how one should goes about ensuring that the certificate of the Coq code caries over to the extracted Futhark code, or at least parts of it.

In Section 5 we will introduce the problem of finding the maximum segment sum, as well as describe our implementation and verification of.

In Section 6 we evaluate in the use of MetaCoq and ConCert (through FuCert) to generate certified Futhark code. We do so by evaluating in things discovered during the process of implementing/certifying the maximum segment sum function, as well as creating FuCert infrastructure. Following that, in Section 6.3, we draw our conclusions on the usability of this method of generating certified Futhark code, based on the evaluation.

Now, we have two appendices, namely, Appendix A where we give an overview of the accompanying code, where to find it and how to compile it, and Appendix B, where we present various code listing that might be of interest to the reader.

[2]See [1].

SECTION 2

Futhark

Futhark is a small functional programming language, developed at the Department of Computer Science at University of Copenhagen. It is a purely functional, data-parallel programming language meant for creating highly efficient code for GPU execution, though it it also has a C backend for CPU execution. It is not meant as a general purpose programming language, but rather a language used for specific problems that are performance-sensitive, possibly by calling compiled Futhark programs from applications written in other programming language. Futhark is designed so that the (ahead-of-time) compiler can make use of very aggressive optimisations, in order to achieve highly data-parallel code. Because of this, the design puts certain limitations on the user.

We will not go into much details about the Futhark language, but rather try to give the reader a feel of what type of language it is, and then discuss some of the points that are highly relevant to this project.

Futhark is a statically typed array based language. The basic types and Futhark are booleans and various integer and floating point values. With regards to type constructors, it, for example, has product types and function types, but, most importantly, it has array types. These array types have a size parameter, which means that Futharks has a light form of dependent types. We will not say much more about the Futhark type system, though there is much more to it, but mention that it also features parametric polymorphism.

An example of a Futhark function definition include the following, which is taken from the front page of the Futhark language homepage,[3]. The function calculates the average of an array of floating point numbers.

```
let average (xs: []f64) = reduce (+) 0.0 xs / f64.i64 (length xs)
```

First of, the definition states that the function being defined, `average`, takes one argument, `xs`, of type `[]f64`, meaning that it is an array of double precision floats. The body on the function consists of a reduce operations, summing up the element of the array (we will talk more about the `reduce` operator later), whose result is divided by the length of the input array, converted from a 64-bit integer to a double precision float before the division.

We will not go further into details with the Futhark language, as we do not need it, but will instead focus on a few points that are of high relevance, namely, which aspects might be relevant to certify in Futhark. If the reader is interested in a thorough introduction to Futhark, he or she may consult [8].

2.1 Array combinators

Futhark has a number of arrays combinators, meaning, operations for performing bulk operations on arrays. It divides these combinators into two categories, *first-order array*

[3] `https://futhark-lang.org/`

combinators and *second-order array combinators* (*SOACs* for short). The latter is simply the combinators that take a function argument, the distinction being that first-order array combinators always do the same whereas the operations the second-order array combinators perform depend on a function argument.

We will not say much about the first order array combinators, many of which do what they are expected to do. These could be `concat`, `replicate`, `transpose`, `zip` and `unzip`. We will briefly mention that the `iota` function takes an integer argument, `n`, and returns an integer array of length n containing the numbers from 0 to $n-1$.

Now, when it comes to second-order array combinators, we will focus on a few of these. Four of the combinators (with their type annotation) are:[4]

$$\begin{aligned}
\texttt{filter} &: \forall\alpha.\,(\alpha \to \texttt{bool}) \to \Pi n.\,\texttt{[}n\texttt{]}\alpha \to \texttt{[}n\texttt{]}\alpha \\
\texttt{map} &: \forall\alpha\,\beta.\,(\alpha \to \beta) \to \Pi n.\,\texttt{[}n\texttt{]}\alpha \to \texttt{[}n\texttt{]}\beta \\
\texttt{reduce} &: \forall\alpha.\,(\alpha \to \alpha \to \alpha) \to \alpha \to \Pi n.\,\texttt{[}n\texttt{]}\alpha \to \alpha \\
\texttt{scan} &: \forall\alpha.\,(\alpha \to \alpha \to \alpha) \to \alpha \to \Pi n.\,\texttt{[}n\texttt{]}\alpha \to \texttt{[}n\texttt{]}\alpha
\end{aligned}$$

The `filter` and `map` operators are, of course, very well known and behave as they are expected to do. The `reduce` operation is also well known, as it behaves like the usual fold operations. However, in order to allow parallel execution, the function puts some less common restrictions on its arguments, which we will expand upon in Section 2.2. The last of the operators might be the least common. The `scan` operation returns an array of partial results of a fold left operation, or, in other words, the `reduce` operation of increasing prefixes of the array. This latter description is also the reason it is called the *generalized prefix sum*. Since we require the result to be an array of the same length, there are two natural choices of definition, namely, the one where the first element of

$$\texttt{scan}\ \odot\ e\ \texttt{[}a_1, a_2, \ldots, a_n\texttt{]}$$

is e and the last is $e \odot a_1 \odot \ldots \odot a_{n-1}$, corresponding to the reduce operations of the prefixes `[]`, $\texttt{[}a_1\texttt{]}, \ldots, \texttt{[}a_1, a_2, \ldots, a_{n-1}\texttt{]}$, and the one where the first element is $e \odot a_1$ and the last one is $e \odot a_1 \odot \ldots \odot a_n$, corresponding to the reduce operations of the prefixes $\texttt{[}a_1\texttt{]}, \texttt{[}a_2\texttt{]}, \ldots, \texttt{[}a_1, a_2, \ldots, a_n\texttt{]}$. The former version is called the *exclusive* scan, and the latter is called the *inclusive* scan. The `scan` operation in Futhark is *inclusive*. Now, like the `reduce` operation, the `scan` operation also has some unusual requirements on the arguments, which we will discuss in the following section.

2.2 The monoid criteria

As mentioned, the purpose of Futhark is to be able to generate highly data-parallel programs, but, a priory, the reduce and scan operations seem highly sequential. For this reason the function supplied to `reduce` and `scan` must be associative, and the supplied

[4]Here we have used the notation of [10] where the placement of the Πn indicates where the size parameter becomes fixed, in case of a partial application.

element must act as a neutral element with respect to this operation. In other words, for the calls

$$\texttt{scan}\ f\ e\ a \qquad \text{and} \qquad \texttt{reduce}\ f\ e\ a$$

to be legal, the function f must make the domain into a monoid with neutral element e. Before we explain in detail what this monoid criteria looks like, let us explain why this is desirable. Supposing that we have two non-empty arrays, $a = [a_1, \ldots, a_n]$ and $b = [b_1, \ldots, b_m]$, then, because of the restrictions,[5]

$$\begin{aligned}\texttt{reduce}\ \odot\ e\ (a +\!\!+\ b) &= e \odot a_1 \odot \ldots \odot a_n \odot b_1 \odot \ldots \odot b_m \\ &= (e \odot a_1 \odot \ldots \odot a_n) \odot (e \odot b_1 \odot \ldots \odot b_m) \\ &= (\texttt{reduce}\ \odot\ e\ a) \odot (\texttt{reduce}\ \odot\ e\ b)\end{aligned}$$

We see here that the result of a reduce operation can actually be calculated in parallel. It is easy to check that the same holds if one or more of the arrays are empty, and this simply shows that the function $\texttt{reduce}\ \odot\ e$ is a list homomorphism. Now, the reasons why `scan`benefit from the monoid criteria is a but more complicated.[6]

Now, formally, a monoid is defined as follows:

Definition. Suppose that we are given a set M together with a binary operation $\odot\colon M \times M \to M$ and an element $e \in M$, satisfying

(1) $m \odot e = e \odot m = m$ for all $m \in M$, and

(2) $(m_1 \odot m_2) \odot m_3 = m_1 \odot (m_2 \odot m_3)$, for all $m_1, m_2, m_3 \in M$,

then the triple $(M, \odot, e)$ is called a *monoid*. The element e is called the neutral element, and the second criteria is referred to as the operator being associative.

Let us discuss what exactly is required when using the functions `reduce` and `scan`. Say we have an operator $\odot\colon T \times T \to T$ (where T is a Futhark type), an element $e \in T$ and an array a of type `[]`T. We do not actually need $(T, \odot, e)$ to be a monoid, rather, with E denoting the set of different elements in a, we need the closure[7] of E with respect to the operations $\odot$ to form a monoid with that operation and neutral element e. This is of course a bit technical, but the takeaway is that there might be a subset of T which is a monoid with the given operation, and the operator is only applied to elements of this subset. A quick example could be the operator $x \odot y = \min\{x, y\}$ which, with the set $\{x : x \leq 0\}$, makes a monoid with neutral element 0, while it is not a monoid with all of $\mathbb{Z}$.[8] Later in Section 5, we will see an example where the function is defined on tuples of integers, but where the monoid criteria is not satisfied on the entire product, and we will solve this problem by using subset types is Coq.

[5]Here we have written the result of the reduce operation, for a non-empty array, is $e \odot a_1 \odot \ldots \odot a_n$, but we can dispense with the first e, as it is a neutral element, so that the result is $a_1 \odot \ldots \odot a_n$.

[6]The easiest way of seeing it is probably by the fact that the monoid criteria allows the `scan` operation to be calculated in parallel be using an up-sweep followed by a down-sweep. See [5] for more information.

[7]The closure can be described as the smallest set, X, containing E so that $x \odot y \in X$, for all $x, y \in X$. It can also be thought of as the set obtained by iteratively adding products of elements.

[8]The operator is indeed associative on all of $\mathbb{Z}$, but there is not a neutral element that works on the entire set.

2.3 Properties to certify

There are of course many things that one could verify about Futhark programs that one might want to verify about programs in other programming languages as well, but here we will try to motivate a few, some of which may be of even more interest in the context of Futhark.

2.3.1 The monoid criteria

An obvious candidate to verify/certify is the monoid criteria needed by the `reduce` and `scan` operations. This criteria is obviously not something the compiler can check; both because of the potential mathematical complexity involved and also because the need to "guess/choose" a restricted domain for the function. For this reason, it is, in fact, up to the user to check that this criteria is satisfied.

2.3.2 Functional correctness

Functional correctness if of course always a candidate as something to verify, but in the case of Futhark, it might be the case that it is more likely to be relevant. The reason for this is due to the nature and purpose of Futhark. Because the purpose of Futhark is to be aggressively optimized and compiled to highly parallel code, it has some language limitations, and might force the user to write programs in a style that he/she would not have done otherwise. For this reason it might be less obvious that the program does what is is supposed to, and a formal verification of this might be a good option.

2.3.3 Legal array indexing

Many languages, including Futhark, insert out of bounds checks are every place there array indexing occurs. However, this might produce an overhead that sometimes can be quite significant depending on the task and how often indexing is done. Futhark is certainly a language where this can make a great difference, as it is indeed an array based language, and executes on GPUs, where the overhead can be even greater, due to synchronization overhead between cores and the way conditionals are handled.

Now, ideally one would check this indexing statically, ahead of time, but this is generally not possible. In Futhark, there is a possibility to disable out of bounds checks, with the `#[unsafe]` attribute, in case the programmer is certain that the indexing is correct. This is generally a dangerous thing to do, but in case it is correct, this can remove the overhead introduced by these checks. This is where certification comes into play. Indeed, if the indexing is certified to be within bounds always, we may safely disable the out of bounds check, in order to produce efficient code.

An example of how this can be done in the FuCert framework (that we introduce in 4) has been implemented by D. Anenkov as an example in this framework. The maximum segment sum example that we are going to go through naturally does not include any array indexing, so this is not something relevant to certify for that example.

SECTION 3

Coq

In this section, we discuss a few aspects of the proof assistant Coq that are relevant to the current project. Mostly, this will be aspects that might present problems when working in Coq in connection with the FuCert framework. How these problems relate to FuCert will be discussed in Section 4.3. Most notable is the concept of proof irrelevance and related concepts.

Coq is a proof assistant, or interactive theorem prover, that allows the user to define functions and data structures and prove assertions about these. Proving in Coq is largely done using the notion of tactics. For an introduction to Coq, the reader may consult [6]. We will not go into details with how Coq works, but we will mention that the underlying language of Coq is a *Calculus of Inductive Constructions* (or *CIC* for short).[9]

3.1 Subset types and equality

Coq has great support for subset types, through the type `sig`. As one would expect, `sig` has the type `forall {A : Type} (P : A -> Prop), type`,[10] so that it consists of all elements in a type, `A`, that satisfies some predicate, `P`. Now, `coq` has a single constructor, `exist`, so that elements of `sig P` has the form `exist P a pf`, where `a` has type `A` and `pf` is a proof of `P a`, that is, an element of type `P a`. Now, as these are used a lot, the following notation is used:

`{x : A | EXP}` is notation for `sig (fun x : A => EXP)`,

for a Coq expression `EXP` of type `Prop`, which will usually depend on `x`. An example could be all negative integers, x, for which x^2 is greater than 10, which can be described as:

`{x : Z | x < 0 /\ x*x > 10}` meaning `sig (fun x : Z => x < 0 /\ x*x > 10)`

Now, this way of using subset types in Coq is very flexible, and the user can, at his or her leisure, create subset types corresponding to an arbitrary predicate. However, it comes with some problems, as equality no longer works like one might expect. Let us suppose that we are given a type `A`, a predicate, `P`, on `A`, and two elements `x` and `y` of type `{z : A | P z}`, one might ask whether `x = y`. Usually, one might think of this question as a question about equality between two elements of `A`. However, this is not the case, and the equality is really an equality of the form

```
exist P vx pfx = exist P vy pfy
```

for some values `vx` and `vy` of type `A` and proofs `pfx` and `pfy` of `P x` and `P y`, respectively. But then we see that for the two values `x` and `y` to be equal, we do not only require the

[9] For more about the basis of Coq in CIC, the reader may consult `https://coq.inria.fr/refman/language/cic.html`.

[10] The curly here means that the argument is implicit, as it can be inferred from the predicate.

underlying values of `A` to be equal, but also that the proofs that `x` and `y` satisfy `P` are equal. This latter fact is quite contrary to how subsets are treated in mathematics and how many people might think about subset types.

This problem with equalities might seem like quite a problem, but there are ways to deal with it. One of these is adding the axiom of *proof irrelevance*, which is consistent with the theory of Coq[11] The axiom of proof irrelevance looks as follows:

```
Axiom proof_irrelevance : forall (P : Prop) (p1 p2 : P), p1 = p2.
```

This axiom asserts that there is at most one proof of any proposition, and indeed, it allows us to prove the following lemma:[12]

```
Lemma subset_eq_compat:
  forall (U : Type) (P : U -> Prop) (x y : U) (p : P x) (q : P y),
    x = y -> exist P x p = exist P y q.
```

which shows that equality of subset types behaves as one would think, as in mathematics. As we will see in the next section, there are some cases where we can get some kind of proof irrelevance without assuming this axiom. There is also a more difficult problem that relates to subset types. We will expand upon this in Section 3.3

3.2 Decidable types and proof irrelevance

Regarding the problems of equality of subset types mentioned above, it turns out that, in some cases, we get a type of proof irrelevance for free. To be more precise, for types with decidable equality, we have *uniqueness of identity proofs* (or UIP). This might seem unnecessary, since we can just assume the axiom of proof irrelevance described earlier. However, there is some merit to not assuming axioms if one can avoid it, even if these axiom are consistent with Coq (with CIC). The reason for this is that the user might want to assume axioms themselves, and *some of these* might be inconsistent with the axioms we have added.

Let us start by defining what a type with decidable equality is.

Definition. A type `A` is said to be *decidable* if there is a proof of the following proposition:

```
forall x y : A, {x = y} + {x <> y}.
```
(1)

The point of our interest in decidable types is that, for a decidable type `A`, there is only one equality proof, namely `eq_refl`. In Coq terms, we have the following result if `A` is decidable:

```
forall (x y: A) (p1 p2 : x = y), p1 = p2
```

[11] See for example [6, Chapter 12]. The axiom can be added by importing the standard Coq module `ProofIrrelevance`.

[12] This lemma is included in the module `ProofIrrelevance`.

which precisely says that there is only *one* proof of equality between elements of the type `A`; hence the name uniqueness of identity proofs. It turns out to be quite convenient that we can actually get this same result for types that satisfy something weaker than decidability. Indeed, we can restrict ourselves to require that the type `A` satisfies:

```
forall x y : A, x = y \/ x <> y.
```
(2)

This statements obviously seems quite similar to decidability, but where en element of type (1) is a function that takes two elements and returns either a proof of their equality or inequality, an element (2) is a function that takes two arguments and returns a proof that the two elements are either equal or not, without specifying which is the case. The reason that these two notions are not equivalent has to do with how Coq separates programs and proofs with the two universes `Set` and `Prop`, and how we are not allowed to pattern match on elements of type `Prop`, since proofs are not allowed to have any influence on programs in Coq.

In order to make things easier and to automate some of the argument resolution, we have created a type class representing types with this "weaker decidable equality":

```
class Dec (a : Type) : Prop :=
  { dec : forall x y : a, x = y \/ x <> y }.
```

In this way, with the implemented instances, a decidability argument to a function might be resolved automatically. Likewise, we have also created a type class representing subset types for which the equality behaves as described above, but formulated with projections:

```
Class SigEq {A : Type} (P : A -> Prop) : Prop :=
  { subset_eq : forall x y : sig P, proj1_sig x = proj1_sig y -> x = y }.
```

With this notation, the statement that (2) implies uniqueness of identity proofs allows us to supply the instance

```
Instance SigEq_dec {A B : Type} `{Dec B} {f g : A -> B}
  : SigEq (fun x => f x = g x)
```

which is what we really are interested in. This proves that if the predicate of a subset type is an equality of a "decidable type" (in this weaker sense), then it behaves as we would like with respect to equality.

Though we decided to require (2) instead of decidable equality, we are most likely only going to use it for decidable types. However, there is a more important reason for this choice, in addition to the statement being weaker. As it is, `Dec A` lives in `Prop` whereas it would live in `Set` had we required decidable equality. Had `Dec` lived in `Set`, arguments of this type would not be removed during extraction, and we would very much like them to disappear.

Let us end by mentioning that many of the usual types are decidable, such as natural numbers, integers and booleans. Moreover, we prove (it is not that difficult), that lists and products of decidable types are also decidable.

3.3 Equality between equal types

We have seen how equality between subset types might give rise to problems if the axiom of proof irrelevance has not been assumed. In this section we will look at a more serious problem, namely, that in certain cases, one might want to state an equality between elements of, a priori, different types, because you know that the types are equal. Let us illustrate such a situation with an example. Consider the following silly scenario, where we have a series of types, parameterized by the natural numbers, each of which is a singleton type (meaning it only has a single element).

```
Inductive Nat : nat -> Type :=
  | N : forall n, Nat n.
Lemma silly: forall (n m : nat) (x : Nat n) (y : Nat m),
    n = m -> x = y.
```

The result seems very obvious; the premise is that `n = m`, so that the elements actually belong to the same type, and as that type only has a single element, the two element must be equal. There is just one problem. This is not a valid Coq program, that is, the statement of the lemma is not a valid CIC term. The problem is that, from Coq's perspective, `x` and `y` have different types, `Nat n` and `Nat m`, and as the definitions of equality (with `=` as notation) is given by:

```
Inductive eq (A : Type) (x : A) : A -> Prop := eq_refl : x = x
```

both arguments must be the same type. The problem is not directly related to subset types, or even dependent types. We could have given the (also rather silly) example term:

```
forall (A B : Type) (x : A) (y : B), A = B -> x = y.
```

which is a convoluted way of stating commutativity of equality. This example is probably more esoteric than the other, and the scenarios where this example is most likely to come up is probably for dependent types; cases where the dependent values are shown to be equal. Later, in Section 4.3, we will see that a place where this causes problems is when comparing vectors of equal lengths.

A way around this problem involves working with *heterogeneous equality*, namely, the alternative equality predicate, `JMeq`,[13] defined as

```
Inductive JMeq {A : Type} (x : A) : forall {B : Type}, B -> Prop :=
  JMeq_refl : JMeq x x.
```

The difference here is, of course, that the two arguments of the equality can have different types, though a proof of `x = y` can exist if `x` and `y` are equal, and, in particular, have the same type. The hope would then be that one would be able to prove the result

```
Theorem JMeq_eq : forall (A : Type) (x y : A), JMeq x y -> x = y.
```

[13] Also referred to as *John Major equality*.

However, this theorem is not provable within CIC, although it seems like it ought to be. Fortunately, it is consistent with CIC (see [6, Section 10.4]), and is often assumed as an axiom.

In our work, we have not assumed that heterogeneous equality implies normal equality. Instead, we can, most of the time, make do with the following result we have proved.

```
Lemma heterog_subset_eq_f {D X Y : Type} {P : D -> X -> Prop}:
    forall (sigeq : forall (d : D), SigEq (P d))
      (f : forall {d : D}, sig (P d) -> Y)
      (d1 d2 : D) (x1 : sig (P d1)) (x2 : sig (P d2)),
    d1 = d2 -> proj1_sig x1 = proj1_sig x2 -> f x1 = f x2.
```

At first, this lemma looks quite complicated, so let us explain what is going on. We are considering elements of a subset type `sig (P d)`, dependent on a parameter `d` of type `D`, and from `sigeq`, we know that all these subset types behave as we would like with regards to equality. Then we have parameters `d1` and `d2` that we know are actually equal, and elements `x1` and `x2` of types `sig (P d1)` and `sig (P d2)`. Now, morally, `sigeq` tells us that two elements of `P d`, for some `d` are equal if their projections are equal, so as the projections of `x1` and `x2` are equal, and their types are also equal, since `d1 = d2`, they should morally be equal. However, because of the subtleties mentioned above, we cannot state their equality, but what we can do is to state that every function, `f`, will take the same value on `x1` and `x2`, as long as the type of the codomain of `f` does not depend on the dependent on `d1` and `d2`.

Now, this might seem like a small thing, but for many purposes this property will turn out to be essential.

3.4 Equality of functions

The last problem/complication in Coq that we will discuss regards equality between functions. In mathematics, function equality means that two functions agree on all elements of the domain. However, in Coq, this is not the case. Like all other terms, functions are only equal if they are *definitionally equal* or *convertible* as terms.[14] This means that

```
Lemma funs1: (fun n : nat => 1 + n) = (fun n : nat => n + 1).
```

is unprovable, since the terms on the two sides of the inequality are not convertible, whereas we are easily prove the following lemma

```
Lemma funs2: (fun n : nat => S n) = (fun n : nat => if true then 1 + n else 0).
Proof. reflexivity. Qed.
```

There is not much to do about this, besides adding the axiom of functional extensionality, that is,

[14] We will not go into details with what definitional equality means, but just briefly mention that two CIC terms are definitionally equal if they reduce to the same normal form using a certain set of reductions. For more information, the reader may consult [6, Chapter 10] or the Futhark manual: `https://coq.inria.fr/refman/language/core/conversion.html`.

```
Axiom functional_extensionality_dep : forall {A} {B : A -> Type},
  forall (f g : forall x : A, B x),
  (forall x, f x = g x) -> f = g.
```

This just says that functions are equal if they agree on all elements of the domain, allowing for a dependent co-domain. We will not say too much more about functional extensionality, as it does not pose much of a problem for this project. Indeed, it seems like results that one might be inclined to prove using the FuCert framework will not find this to be a problem. Often there is no reason to state an equality as an equality of functions when one can state it as a quantified equality of applied functions.

SECTION 4

The FuCert framework

The FuCert framework is a framework in Coq for extracting Coq code to Futhark. It is based on the ConCert framework, which, in turn, is build on the MetaCoq framework.

In this section, we will give a brief sketch of how these two latter frameworks work, and how FuCert employs them. Afterwards we will go through some of the parts of the project that developed new FuCert infrastructure. We will do so by first discussing how one can meaningfully choose representatives for Futhark functions for extraction, then afterwards discussing how Futhark arrays can be modelled in Coq, and lastly discuss what part of the Futhark infrastructure was implemented in this project.

4.1 MetaCoq, ConCert and FuCert

As the name suggests, the MetaCoq project[15] is a metatheory of Coq, which in particular provides tools for manipulating Coq terms in Coq. One thing the project has accomplished is to reimplement the extraction feature of Coq in Coq (see [12]). Where the extraction feature of Coq extracts to OCaml, Haskell or Scheme, MetaCoq's version of this erasure procedure targets a the language $\lambda_{\square}$. This language, introduced by MetaCoq, is an untyped lambda calculus, with a particular constructor $\square$, representing everything that does not have any computational relevance.

The ConCert project[16], whose main purpose is smart contract verification, has an erasure procedure for types (see [2]), which allows them to augment the $\lambda_{\square}$ term resulting from MetaCoq's erasure procedure with additional type information. This augmented version of $\lambda_{\square}$, denoted $\lambda_{\square}^{T}$, then contains enough typing information to be able to extract to many Hindley-Milner based functional programming languages.

From the $\lambda_{\square}^{T}$ representation, a person simply needs to write a pretty printer in order to extract to a given target language. The FuCert framework utilizes this augmented erasure procedure from ConCert to extract Futhark code, that is, the FuCert framework has a pretty printer from $\lambda_{\square}^{T}$to Futhark code. This printing takes a set of remappings

[15] https://github.com/MetaCoq/metacoq
[16] https://github.com/AU-COBRA/ConCert

that the pretty printer uses to print certain functions and types, as well as a preamble that is inserted in the beginning of the Futhark program.

In the next section, we will discuss what role the remapping mentioned above has in certifications.

4.2 Remapping functions

In order to write Coq programs that exports to Futhark code in a meaningful way, the user will need representations of the Futhark operations in Coq, so that these functions can be remapped to the corresponding Futhark operations. Otherwise, the Futhark program would include its own implementation of operators such as `map`, `filter`, `reduce` and `scan`, which would not only be silly, but also rob the user of many of the optimisations the Futhark compiler makes, in order to create efficient parallel code.

At this point a problem presents itself. When remapping certain functions during extractions, there are no verifications of this being a meaningful remapping. In order for the guarantees made in Coq to also hold for the exported Futhark code,[17] it is paramount that the remapped Futhark functions share the same properties as the original Coq functions. If the proofs, for example, relied on properties of the Coq implementation of a Futhark operator that the Futhark counterpart does not share, then this guarantee does not transfer to the extracted Futhark program. If on the other hand the Coq function is simply missing some of the properties of the corresponding Futhark function, then the user will (possibly) be unable to prove the desired result.

There are two aspects to this problem. One is that the user is unrestricted in his or her remapping, so that he/she can (possibly by mistake) do some sort of silly remapping. This can result in something that does not compile, or even worse, something that does compile but does not produce the expected result. The other aspect is that the user might do a correct remapping, but where it is unclear whether the properties of the Coq function are the same as that of the Futhark function it is remapped to.

In order to combat the former of these two aspects, some control of the remapping would have to be moved from the user to the FuCert frameworks. We have not dealt with this aspect in this project. In order to combat the latter of the two aspects, we have made some infrastructure that the user can use to be guaranteed that the Coq version of the Futhark functions only uses properties that are shared by the Futhark counterparts.[18] Let us explain how we have managed to deal with this aspect.

Suppose (just for the sake of demonstration) that `abs` is the Futhark function we want to write a Coq version of (for calculating an absolute value), and we want to implement it for integers. Then what we have done is to create a module type from the specification of the Futhark functions (in this case just one), namely,

```
Module Type FutharkSpec.
  Variable abs : Z -> Z.
```

[17] There are many subtleties in what guarantees one get for exported code, and we will talk more about that in Section 5.6.

[18] As we will discuss later, we have only implemented a small part of the Futhark functions so far, but the method works in general, and the implementation can be extended.

```
  Axiom abs_pos: forall x : Z, 0 <= x -> abs x = x.
  Axiom abs_neg: forall x : Z, x <= 0 -> abs x = -x.
End FutharkSpec.
```

A module implementing this type will then have an `abs` function and satisfy the two axioms, which we have chosen to be the guarantees the actual Futhark function provides. When importing a module of this type, the user then has the `abs` functions available, and it will satisfy the properties the specification states, which is supposed to capture the guarantees the Futhark functions gives. We have then, furthermore, created a functor (a module with parameters), with results derived from these axioms,[19]

```
Module FutharkMod (Spec : FutharkSpec).
  Include Spec.
  Theorem abs_ge0 : forall x : Z, 0 <= abs x.              Proof. ... Qed.
  Theorem abs_idem: forall x : Z, abs (abs x) = abs x.     Proof. ... Qed.
End FutharkMod.
```

In this way, the user is guaranteed that, if he or she imports an application of this module, the results provided about `abs` are guaranteed to be shared by the Futhark versions of the function. The only thing left is to give an actual implementation of the `FutharkSpec` module and make sure that the properties in this module (and consequences hereof) are then only properties used about the function. This is very easily done by making the implementation of the module opaque.[20]

```
Module FutharkSpecImpl : FutharkSpec.
  Definition abs (x : Z) : Z := if 0 <=? x then x else -x.
  Theorem abs_pos: forall x : Z, 0 <= x -> abs x = x.      Proof. ... Qed.
  Theorem abs_neg: forall x : Z, x <= 0 -> abs x = -x.     Proof. ... Qed.
End FutharkSpecImpl.

Module FutharkImpl := FutharkMod (FutharkSpecImpl).
```

Since the implementation, `FutharkSpecImpl`, of `FutharkSpec` is opaque, the actual implementation is not visible. In particular, the user cannot make use of the actual implementation, but will have to rely on the properties the module type `FutharkSpec` exhibits. In short, the user can import the module `FutharkImpl` and get all the properties of the implementation, but not see the implementation details.

Note that `abs` was just an example, and we shall see soon that we have implemented some of the second order array combinators in the actual modules in FuCert of the names given above.

Now, the above solution with a module type and an opaque implementation seem to do a good job of removing the problems concerning remapping. However, it is unfortunately not so simple, and a good example of this is that in order to export to Futhark, we also need to remap types. Types are a bit more complicated for several reasons. Firstly,

[19] We have omitted the proofs for brevity, but they are all straightforward.

[20] For information on the module system in Coq, including opaque modules, the reader may consult `https://coq.inria.fr/refman/language/core/modules.html`.

we usually do not define the basic types ourselves, but would rather work with the ones that are already present in Coq. We could define types that behaved like the Futhark types. Even worse however, we might find that the results we are interested in might not even hold if we faithfully model Futhark types. A good example of this is that addition is not associative on bounded precision integers. However, often the properties we are interested in hold when no overflow is encountered. We will not say more about this for now, as we will take it up again during the discussion in Section 5.6.

4.3 Array model

The first question that pops up when writing Coq versions of the Futhark functions is how one should model Futhark arrays. The easiest would naturally be lists, but this is only practical for very simple programs, where we do not need to do any reasoning that involves array sizes. In particular, we will encounter problems with the Futhark functions that already have restrictions on the array sizes. An example of such a function could be `zip`, which must be called on arrays of the same size.

There are several choices of models for the arrays that capture array sizes, which is the primary interest. We have chosen to use subset types (see Section 3.1). More precisely, given a type `A` and an integer `n`, we use the type

```
{x : list A | length x = n}      (denoted [|n|]A)
```
(3)

to model a Futhark array with length n. But before we elaborate on this choice, let us briefly discuss the alternatives.

One choice of array model could, obviously, be to define a custom datatype resembling lists, while having the length be part of the type argument, such as

```
Inductive FuArr {A : Set} : nat -> Set :=
| FuNil : FuArr 0
| FuCons : forall (h : A) {n : nat} (t : FuArr n), FuArr (S n).
```

With this model, the `zip` function could be defined with the type

```
zip : forall (A : Set) (n : nat), FuArr A n -> FuArr A n -> FuArr (A * A) n
```

This is similar to the solution with subset types that we have used, with the difference that we do not use the `list` type as the underlying data structure, and that we do not carry proofs around.

Another alternative model for the Futhark arrays would be to use pull or push arrays, the latter of which are introduced by Claessen, Sheeran and Svensson in [7]. A pull array is a function from the set of indices to the values of the array element type. This model is quite different from the previous ones, in particular because it is not based on an inductive data type. With this model, we would define the array type, let us denote it `FuArr A n` again, by

```
Definition FuArr (A : Type) (n : nat) : Type := {k : nat | k < n} -> A.
```

This representation seems to be a quite good choice for an array model as it works well with Futhark's approach to arrays as read only, and it also automatically forces array indexing to be safe. This representation also makes some functions easier to define, as they can be described via function compositions, such as:[21]

```
Definition iota (n : nat) : FuArr nat n := @proj1_sig nat _.

Definition map_fu {A B : Type} {n : nat} (f : A -> B) (xs : FuArr A n)
  : FuArr B n := compose f xs.

Definition zip {A B : Type} {n : nat} (xs : FuArr A n) (ys : FuArr B n)
  : FuArr (A * B) n := fun i => (xs i, ys i).

Definition unzip {A B : Type} {n : nat} (es : FuArr (A * B) n)
  : FuArr A n * FuArr B n := (compose fst es, compose snd es).

Definition transpose {A : Type} {n m : nat} (mat : FuArr (FuArr A m) n)
  : FuArr (FuArr A n) m := fun i j => mat j i.
```

However, this comes with a price, namely, that many things become significantly more complicated to define and prove. The main reason is that there is no natural candidate for an induction principle, since there is no underlying inductive data structure; the primary problem being the induction step. One candidate for the induction step could be

```
forall (n : nat), (forall ys : FuArr A n, P ys)
  -> forall (xs : FuArr A (S n)) (h : A),
       xs (exist _ 0 (lt_0_succ n)) = h -> P xs.
```

but it is unlikely to be easy to apply, because the conclusion, `P xs` is not formulated in terms of an array of size n and a value in `A`, so it will be difficult to apply the assumption `forall ys : FuArr A n, P ys`. We will see in a moment, when discussing the model using subset types, that we will have to do some work to deal with equality, and the same will be the case for pull arrays. In order to make arrays with identical elements be equal, we would have to assume the functional extensionality axiom (see Section 3.4).

Let us talk a bit about the model with subset types, (3), that we have selected (we will refer to this as the *array type*), as well as some of the implementations we have made to support its usage. To a large degree, all the definitions and results are adaptations of the usual results for lists, adapted to array types, but working around the complications mentioned in Section 3.1, regarding proof irrelevance, as well as those mentions in Section 3.3, related to heterogeneous equality. It is worth noting that the problems related to heterogeneous equality are not specific to the representation we have chosen, but will be a problem for every model that has the array type being dependent on the array length. Moreover, as the representation of Futhark types is most likely going

[21] We have included all these to illustrate the ease. The absence of the `reduce` and `scan` functions does not indicate that these are difficult; they are also quite easy, but do not only involve function composition.

to be types with decidable equality, we get proof irrelevance for free, as mentioned in Section 3.1. However, it still complicates the setup, as proof irrelevance is not build into Coq, but rather an assumed axiom.

In order to be able to work comfortably with array types, the first relevant thing to do is to define cons and append operators, similar to the versions for lists, `::` and `++`. Defining versions of those for the array type is quite straightforward, and just applies the list version on the element underneath and updates the proof argument accordingly. The operators corresponding to `::` and `++` are denoted `[::]` and `[++]`, respectively. With these operators defined, especially the cons operator, we made a destruction principle and an induction principle for the array type, so as to be able to more easily do induction proofs. The induction principle, `arr_ind`, (with the proofs removed for brevity) looks as follows:[22]

```
Context {A : Type} `{Dec A}.

Variable P          : forall {n : nat} (arr : [|n|]A), Prop.
Hypothesis base_case : P (exist _ [] eq_refl)
Hypothesis ind_step  : forall (n : nat) (a : A) (arr : [|n|]A),
  P arr -> P (a [::] arr).

Lemma arr_ind        : forall (n : nat) (arr : [|n|]A), P arr.
```

so that we, for an array `xs` of type `[|n|]A`, can use this principle for induction with `induction n, xs using arr_ind`. Let us explain how this works. When doing induction, the variable `P` is the proposition you are proving, which depends on an array, `arr`, and therefore also on its size, `n`. Thus the conclusions of `arr_ind` is exactly the statement you want to prove, and the result says that it suffices to prove the results `base_case` and `ind_step`. These are obviously the statements that correspond to the base cases and the induction step.

This induction will, to a high degree, allow us to do proofs without having to destruct the subset types. This makes proofs more clean and removes some of the tediousness that the need for proof irrelevance brings.

Besides the induction principle, we also have various other results making it easier to work with this array type, in particular, a `destruct` principle, `arr_dest`, so that the user can do cases analysis on arrays by proving the statement for the empty array and for an array with the `[::]` cons operator. The statement of `arr_dest` is the same (up to renaming) as `arr_ind` with the premise `P arr` removed in `ind_step`. One of the other results is an induction principle for non-empty arrays, `arr_ind_S`, as well as a destruction principle for non-empty lists. The latter is something that does not have an analogue for lists, but as we know that an array of length `S n` is non-empty, we must be able to write it as `h [::] t` for some element `h` and array `t` of length n.

We will not go further into details with the various results that make working with arrays easier, but instead skip ahead to the parts that has to do with array content. More precisely, we have defined three propositions, one representing that an array has

[22] For the meaning of `Dec A`, see Section 3.2.

a certain element at a given index, one representing that an array is a prefix of another array, and one representing that an array is a segment of another array, where a segment is a contiguous set of elements of the array.[23] Let us start with the index statement, but note that, for all three definitions, we have the context `{A : Type} `{Dec A}`. The proposition `Index i a xs` should be understood as `x` being at position `i` in the array `xs`, and it is defined inductively as:

```
Inductive Index : forall {n : nat} (i : nat) (a : A) (xs : [|n|]A), Prop :=
| IndexHead : forall {k : nat} (a : A) (t : [|k|]A), Index 0 a (a [::] t)
| IndexTail : forall {k : nat} (i : nat) (a h : A) (t : [|k|]A),
    Index i a t -> Index (S i) a (h [::] t).
```

It should be clear that this gives the intended semantics to the statement, as an element in the array is either the head, being at index `0`, or in the tail of the array, at some index `i`. In the latter case this means that the element is at index `S i` in the original list.

The proposition about prefixes is defined in a similar manner. The proposition `Prefix p l` should be understood as the array `p` being a prefix of the array `l`; it is defined as follows.[24]

```
Inductive Prefix : forall {l n : nat} (p : [|l|]A) (xs : [|n|]A), Prop :=
| PrefixEmpty : forall {k : nat} (xs : [|k|]A), Prefix nil_arr xs
| PrefixHead  : forall {l k : nat} (h : A) (p : [|l|]A) (xs : [|k|]A),
                    Prefix p xs -> Prefix (h [::] p) (h [::] xs).
```

Like with `Index`, it is easy to see that `Prefix` has the described semantics; either the prefix is empty, or it shares the head with the other array and the tail of the prefix is a prefix of the tail of the other array.

Last we have the proposition about segments, where `Segment l1 l2` should be understood as `l1` being a segment in `l2`. It is defined by

```
Inductive Segment : forall {n1 n2 : nat}, ([|n1|]A) -> ([|n2|]A) -> Prop :=
| SegmentHead : forall {n1 n2 : nat} (l1 : [|n1|]A) (l2 : [|n2|]A),
    Prefix l1 l2 -> Segment l1 l2
| SegmentInner : forall {n1 n2 : nat} (h : A) (l1 : [|n1|]A) (l2 : [|n2|]A),
    Segment l1 l2 -> Segment l1 (h [::] l2).
```

so that `l1` is a segment in `l2` if it is either a prefix or a segment in the tail of `l2`. The notion of segment might seem a little less natural to include than the other two, but we will need it in Section 5. We have also included some further results involving these three definitions, but we will not expand on this. Instead we will move on to the Futhark infrastructure.

4.4 Futhark infrastructure

Let us describe the Futhark infrastructure we have implemented during this project, that is, we will describe the modules `FutharkSpec`, `FutharkMod` and `FutharkSpecImpl` mentioned

[23] We will talk more about segments in Section 5, but the reader may consult Definition 5.

[24] Here `nil_arr` defined to be the empty array, that is, defined as `exist _ [] eq_refl`.

in Section 4.2. We will only concentrate on the implementation of the `scan` and `reduce` functions, as these are the most interesting. However, the modules also contain specifications, implementations and auxiliary results for the `map`, `zip` and `unzip` operators.

Now, before going into the details of this, let us explain how the monoid criteria is dealt with (see Section 2.2). We do this in the same way as Annenkov has done in the existing examples in the FuCert framework, that is, with the type class:

```
Class IsMonoid (M : Type) (op : M -> M -> M) (e : M) : Prop :=
  { munit_left  : forall m, (op e m) = m;
    munit_right : forall m, (op m e) = m;
    massoc      : forall m1 m2 m3, op m1 (op m2 m3) = op (op m1 m2) m3
  }.
```

With this definition, the SOACs that require the monoid criteria to be satisfied can simply take an `IsMonoid` argument, and a type class is used, which can automatically be resolved.

At this point the choice of signatures for `scan` and `reduce` in the module type (or interface), `FutharkSpec`, are straightforward, namely,

```
Parameter reduce:
  forall {A : Type} `{Dec A} (op : A -> A -> A) (ne : A) `{IsMonoid A op ne}
    {n : nat} (xs : [|n|]A), A.
```

and

```
Parameter scan:
  forall {A : Type} `{Dec A} (op : A -> A -> A) (ne : A) `{IsMonoid A op ne}
    {n : nat} (xs : [|n|]A), [|n|]A.
```

If we take a look at the explicit arguments, then we see that both functions take an operation `op`, an element `ne` and an array `xs`, and then return a value of type `A` (for `reduce`) or `[|n|]A` (for `scan`). Among the implicit arguments we have the type and the array length as these can be inferred, but we also have two type classes that ensure that `A` has decidable equality and that it forms a monoid with the operation `op` and neutral element `ne`. The reason for having as many arguments as possible being implicit is obviously to make the usage more smooth, and to make the type signature look like the Futhark counterparts.

Now, besides these parameters, the module type also need to contain axioms that describe the properties of the Futhark functions without adding more properties. As mentioned in Section 2.1,[25] the SOACs `map`, `reduce`and `scan`have the following semantics

$$\begin{aligned}
\texttt{map}\ f\ [a_1, a_2, \ldots, a_n] &= [f\ a_1, f\ a_2, \ldots, f\ a_n] \\
\texttt{reduce}\ \odot\ e\ [a_1, a_2, \ldots, a_n] &= a_1 \odot \ldots \odot a_n \\
\texttt{scan}\ \odot\ e\ [a_1, a_2, \ldots, a_n] &= [a_1, a_1 \odot a_2, \ldots, a_1 \odot \ldots \odot a_n]
\end{aligned}$$

[25] We did not explicitly state the semantics of `map`, but only that it is the standard semantics.

given a map f, an associative operator $\odot$ whose neutral element is e, and an array $[a_1, a_2, \ldots, a_n]$. For empty arrays, `map` and `scan` returns empty arrays, and `reduce` returns e. In particular, it is not too difficult to see that, for non-empty arrays,

$$\texttt{map}\ f\ [a_1, a_2, \ldots, a_n] = f\ a_1 :: [f\ a_2, \ldots, f\ a_n] \tag{4}$$

$$\texttt{reduce}\ \odot\ e\ [a_1, a_2, \ldots, a_n] = a_1 \odot (\texttt{reduce}\ \odot\ e\ [a_2, \ldots, a_n]) \tag{5}$$

$$\texttt{scan}\ \odot\ e\ [a_1, a_2, \ldots, a_n] = a_1 :: (\texttt{map}\ (a_1 \odot)\ (\texttt{scan}\ \odot\ e\ [a_2, \ldots, a_n])) \tag{6}$$

Now, we will not discuss these in details, as is is rather elementary, but just mention that it is important that the operator $\odot$ is associative and that e is a neutral element. These properties are the ones we add to the `FutharkSpec` module, so that for `scan` we add the axiom

```
Section scan_axioms.

  Context {A : Type} `{Dec A} (op : A -> A -> A) (ne : A) `{IsMonoid A op ne}.

  Axiom scan_cons:
    forall (n : nat) (h : A) (t : [|n|]A),
      scan op ne (h [::] t) = h [::] map (op h) (scan op ne t).

End scan_axioms.
```

It turns out that this is sufficient to describe exactly the scan semantics, and indeed, in the `FutharkMod` functor, we prove that a `scan` function satisfying this (together with a correct `reduce`) also satisfies

```
Theorem scan_index {i n : nat}:
  forall (p : [|S i|]A) (xs : [|n|]A),
    Prefix p xs -> Index i (reduce op ne p) (scan op ne xs).
```

with the same `Context` as above. This exactly shows us that the element at the k'th index (the $k+1$'th element) is the `reduce op ne p` for the prefix of length $k+1$, exactly as in the description of (6).

Now, for `reduce`, we cannot make do with (5), as we also need to know that it maps the empty list to e. At least this fact does seem likely to be provable from (5), and as we are not necessarily going for a minimum number of axioms (in the sense of excluding those deducible from others), we have included it as an axiom. In other words, the reduce axioms included in `FutharkSpec` are

```
Section reduce_axioms.

  Context {A : Type} `{Dec A} (op : A -> A -> A) (ne : A) `{IsMonoid A op ne}.

  Axiom reduce_nil:
    forall l : [|0|]A, reduce op ne l = ne.

  Axiom reduce_cons:
```

```
  forall (n : nat) (a : A) (l : [|n|]A),
    reduce op ne (a [::] l) = op a (reduce op ne l).

End reduce_axioms.
```

Now, we will not go further into details with the implementation of the infrastructure, but end this section mentioning that we have also implemented the `segm_scan` function from [9]. This is done in `FutharkMod` as it is entirely composed of other functions. From function one can continue on an define more functions from [9], in particular, the `expand` function that allows for solving irregular parallel problems.

SECTION 5

Maximum segment sum

In this section we will describe the example implementation we have done in the FuCert framework, namely, of the implementation of a solutions to the maximum segment sum problem. The implementation was guided by the Futhark implementation of the algorithm, found at the Futhark homepage (see Section B.1 in the appendix).

We will start by presenting the maximum segment sum problem and algorithms for solving it. In particular, we will explain the intuition behind the algorithm used in the example implementation. After that, we will have a look at how this what implemented in Coq, for Futhark extraction, and what verification was done. This verification covered associativity of the chosen operator and functional correctness of the final function. In Section 5.6, we will elaborate on what this certification means for the extracted Futhark code.

Informally, calculating the maximum segment sum means calculating the maximal sum of a contiguous sequence of elements of an array of numbers., but in order to give a formal definition, let us start by introducing a bit of terminology that will be central to this problem. In the following, $\text{len}(A)$ denotes the length of an array A.

Definition. A *segment* (or *sub-array*) of an array A is an array consisting of contiguous elements from A. In other words, S is a segment of A if there exists a non-negative integer n so that $\text{len}(S) + n \leq \text{len}(A)$ and $S[i] = A[i + n]$, for all i with $0 \leq i < \text{len}(S)$.

The segment S of A is said to be a *initial segment* and a *closing segment* if $n = 0$ and $\text{len}(S) + n = \text{len}(A)$, respectively. An initial segment is also called a *prefix*.

It is important to note that the notion of a segment (or sub-array) is different from a subsequence, where the elements can be spaces arbitrarily in the array, so that $[1, 3, 5]$ is a subsequence of $[1, 2, 3, 4, 5]$, but not a segment. Moreover, with this definition the empty array is a segment in all arrays.

5.1 The problem

There are a few variations of the problem, but we will be concerned with the following.

Given an array of numbers, find the maximal sum of a segment of that array.

We call this sum the *maximum segment sum*. With our definition of segment, we allow for empty segments, but there are also variants that do not allow this. In our case, the maximum segment sum is always non-negative, as we can just take the empty segment. However, if we disallow empty segment, then the maximum segment sum in an array of negative numbers will be the maximum of these numbers.

Another variant of the problem involves finding the indices of the segment, that is, the place in the array where our segment starts and stops. For this variant.

The problem of finding the maximum segment sum was first considered by U. Grenander, and an $O(n)$ algorithm was found by J. Kadane, namely the one presented in Listing 1. See [4].

```
MaxSoFar := 0.0
MaxEndingHere := 0.0
for I := 1 to N do
  MaxEndingHere := max(0.0, MaxEndingHere + X[I])
  MaxSoFar := max(MaxSoFar, MaxEndingHere)
```

Listing 1: Kadane's algorithm, finding the maximum segment sum, `MaxSoFar`, of length `N` array `X`, as presented in [4].

5.2 Kadane's algorithm

Looking at Kadane's algorithm in Listing 1, it is easy to see that this is actually a left fold operation on the array `X`. Indeed, we can easily rewrite is to this, as shown in Listing 2.

```
f :: (Int, Int) -> Int -> (Int, Int)
f (ms, ls) x = let nls = max 0 (ls + x)
               in (max ms nls, nls)

mss :: [Int] -> Int
mss xs = fst $ foldl f (0, 0) xs
```

Listing 2: Kadane's algorithm (Listing 1) as a fold operation in Haskell.

Now, the intuition behind this algorithm is quite simple. We consider increasing prefixes of the array, and we keep track of the current maximum segment sum as well as the largest sum of a closing segment. In that way, we can check if we reach a point where we get a larger maximum segment sum by adding an element at the end of the current array. Let us explain this in a bit more detail as well as justify why this works.

Supposing that we have an array A whose last element is x, and that A' is A with this element removed. Then we can express the maximum segment sum of A in terms of A' and x as follows. Either the maximum segment sum does not change after x is added to the end, or it changes and there is a closing segment with the maximal sum.

Indeed, if the maximum segment sum changes and there is no closing segment with this sum, then there must be a segment in A' with this new sum, contradicting the fact that it changed. Now, let us consider how a closing segment, S, in A with the maximal sum (among closing segments) looks. There are two cases, either the segment is empty, and the sum is zero, or it consists of a closing segment, S', in A' with x added in the end. As the sum of S is the sum of S' plus x, it is easy to see that S' must have the maximal sum among closing segments in A'. To summarize, if ls and ls' are the maximal sums of closing segments in A and A', respectively, and ms and ms' are the maximum segment sums of A and A', respectively, then

$$ls = \max\{0, ls' + x\} \qquad \text{and} \qquad ms = \max\{ms', ls\}$$

Looking at Kadane's algorithm in Listing 1 or 2, we can see that this is exactly the way it operates the maximum segment sum and the maximum sum of a closing segment as it iterates through the array from the left.

It is easy to see that this could just as well have been done with a fold right operation.

5.3 A solution for reduce

Recall from Section 2.2 that the `reduce` operator in Futhark places some unchecked restrictions on its arguments, in the sense that, in a call of the type `reduce op ne xs`, the array `xs` should belong to a domain which is a monoid with operation `op` and neutral element `ne`. If we modify the Haskell code in Listing 2 to fit the type of reduce,[26] it is easy to see that `f` does not constitute an associative operator, so this function can not be used to reduce in a similar way; the problem being the asymmetry in the operator. Indeed, it is easy to check that the function, f say, that calculates the maximum segment sum is not a list homomorphism. Indeed, if it was, and $\odot$ with the associated operator, then:

$$1 = f([1,-1,1]) = f([1]) \odot f([-1,1]) = f([1]) \odot f([1,-1]) = f([1,1,-1]) = 2,$$

which is obviously absurd. Luckily there is an associative operator we can use to solve the problem with. Like before, we will have to consider a function that computes more than just the maximum segment sum.

In Kadane's algorithm, we calculate the maximum segment sum and the maximum sum of a closing segment. This time we calculate four values connected with the array,[27] namely,

- the maximum segment sum,
- the maximum sum of a initial segment,

[26] The Futhark `reduce` has type $\forall\alpha.\,(\alpha \to \alpha \to \alpha) \to \alpha \to \texttt{[]}\alpha \to \alpha$, which is the same as the type $\forall\alpha\,(.\,\beta \to \alpha \to \beta) \to \beta \to \texttt{[]}\alpha \to \beta$ of the Haskell `foldl` with $\alpha = \beta$, so this is just done by having `foldl` operator on `map (\x -> (x, x)) xs` instead of `xs` and updating `f` accordingly.

[27] The four different coordinates are usually named `mss`, `mis`, `mcs` and `ts`, for maximum segment sum, maximum initial sum, maximum closing sum and total sum, respectively, in that order.

- the maximum sum of a closing segment,
- the total sum.

If we look at the Futhark implementation (see Section B.1 in the appendix), then we see that the operator, `redOp`, let us denote it by $\odot$, that is used is given by

$$\begin{bmatrix} x_{\text{mss}} \\ x_{\text{mis}} \\ x_{\text{mcs}} \\ x_{\text{ts}} \end{bmatrix} \odot \begin{bmatrix} y_{\text{mss}} \\ y_{\text{mis}} \\ y_{\text{mcs}} \\ y_{\text{ts}} \end{bmatrix} = \begin{bmatrix} \max\{x_{\text{mss}}, y_{\text{mss}}, x_{\text{mcs}} + y_{\text{mis}}\} \\ \max\{x_{\text{mis}}, x_{\text{ts}} + y_{\text{mis}}\} \\ \max\{y_{\text{mcs}}, x_{\text{mcs}} + y_{\text{ts}}\} \\ x_{\text{ts}} + y_{\text{ts}} \end{bmatrix} \tag{7}$$

We will see later, in Section 5.5, that this is actually an associative operator, with the zero vector as neutral element, and that it actually produces the correct result. However, let us discuss the intuition behind it. If x and y are lists that the above values correspond to, then the reasons for the four coordinates are:

- A segment in $x + y$ of maximum sum must either be a segment in x of maximum sum, a segment in y of maximum sum or the concatenation of a closing segment in x of maximum sum and a left bonding segment in y of maximum sum.
- An initial segment in $x + y$ of maximum sum must either be a initial segment in x of maximum sum or the concatenation of x and a initial segment in y of maximum sum.
- A closing segment in $x + y$ of maximum sum must either be a closing segment in y of maximum sum or the concatenation a closing segment in x of maximum sum and y.
- The total sum of $x + y$ is the sum of their individual sums.

We also see in the Futhark implementation of the maximum segment sum, before calling `reduce` that we transform the array into an array of four tuples. Basically we replace each value with a four tuple corresponding to the singleton list for that value, that is, given a number n, we replace it with $(\max\{0, n\}, \max\{0, n\}, \max\{0, n\}, n)$.

5.4 Implementation and associativity

In this section we will talk about the implementation of the maximum segment sum functions using the FuCert framework, so that successful extraction is possible. As the FuCert infrastructure we have implemented (described in Section 4.4) forces functions defined in terms of the provided `reduce` function to make sure that the monoid criteria is satisfied, we will prove associativity of the operator we have described in the process; done through providing an instance of the `IsMonoid` type class.

Now, for certain operations such as addition, it is clear that their domain constitute a monoid with said operation and an obvious neutral element. However, it is certainly not obvious for the operation $\odot$, from (7), even though it seems very reasonable. Indeed,

it does actually *not* make $\mathbb{Z}^4$ into a monoid, as one might think. The problem with the intuition is that we think of lists, but not every element in $\mathbb{Z}^4$ is associated with a list, in the way we interpret the four values (see Section 5.3). It is however the case that the set of elements $(x_{\text{mss}}, x_{\text{mis}}, x_{\text{mcs}}, x_{\text{ts}})$ in $\mathbb{Z}^4$ satisfying

$$x_{\text{mss}} \geq x_{\text{mis}}, \quad x_{\text{mss}} \geq x_{\text{mcs}}, \quad x_{\text{mis}} \geq x_{\text{ts}} \quad x_{\text{mcs}} \geq x_{\text{ts}} \quad x_{\text{mis}} \geq 0 \quad \text{and} \quad x_{\text{mcs}} \geq 0$$

is a monoid with the operation $\odot$, which is not obvious to begin with. It is rather clear that the neutral element should be $(0, 0, 0, 0)$ and expanding on the equations of this being a left and right unit produces the equations above.[28]

It is easy to see that these equations are natural from the perspective of the intuition about lists. That is, that the maximum segment sum must be greater than both the maximum initial segment and closing segment sums, which in turn both must be greater than zero and than the total sum of the array.

Now, this realization of the restricted domain was the first key in order to prove the monoid laws, and, in Coq, we therefore make use of subset types, as with the array type (see Sections 3.1 and 4.3). An initial thought could be that the subset type we are interested in is

```
Definition X : Type :=
  {x : Z * Z * Z * Z |
    let '(mss, mis, mcs, ts) := x in
    mis <= mss /\ mcs <= mss /\ 0 <= mis /\ 0 <= mcs /\ ts <= mis /\ ts <= mcs.
```

However, with this choice, we would be missing proof irrelevance. Luckily, boolean values have decidable equality, so substituting the above predicate in the subset type with

```
(mis <=? mss) && (mcs <=? mss) && (0 <=? mis)
  && (0 <=? mcs) && (ts <=? mis) && (ts <=? mcs) = true
```

we are guaranteed proof irrelevance, as described in Section 3.2. As the domain of the associative operator is a subset type, it is natural to use the `program` tactic in order to define functions using it. This brings an issue with it though, namely that when defining something with the program tactic there is less control of how the functions are defined in reality. Indeed, with the `program` tactic, we just specify how we intuitively want the function to look. The reason this becomes a problem is that the function might very well have a form that the FuCert framework does not support. Now, in our initial onset, this was one of the problems encountered when defining the associative operator as a functions `redOp: X -> X -> X`. Let us illustrate this problem, and the solution we used, with an example. Consider the following function, `f`, defined on subset types:

```
Definition P (x : Z * Z * Z) : Prop :=
  let '(x1, x2, x3) := x in (0 <= x1) /\ (0 <= x2).
```

[28] This does not necessarily mean that all the values satisfying the equations would correspond to a list. It seems possible that this might be the case, but it is not important, and is not something we have investigated.

```
Program Definition f (x : {z : Z * Z * Z | P z}) : {z : Z * Z * Z | P z} :=
  let '(x1, x2, x3) := x in (x2, x1, x3).
Next Obligation. inversion H; lia. Qed.
```

The function seems rather simple, in that it simply permutes the first two of the three coordinates. However, a call to `Print f` will reveal that the function `f` in reality looks a bit more complicated to the human eye.[29] Moreover, if we export it to Futhark code, we get the function:[30]

```
let f (x : sig_ ((i64,i64),i64)) =
  let filtered_var = id x in
  let program_branch_0 = \x1 -> \x2 -> \x3 -> \Heq_x ->  ( (x2, x1), x3) in
  match filtered_var
    case (p, x3) -> (match p case (x1, x2) -> (program_branch_0 x1 x2 x3)) ()
```

which is not valid Futhark since the innermost match case returns a function. This type of problem can be difficult to foresee, and might seem difficult to get around. In our particular case, we can get around it by replacing the definition of `f` with

```
Definition f_aux (x : Z * Z * Z) : Z * Z * Z :=
  let '(x1, x2, x3) := x in
  (x2, x1, x3).

Program Definition f (x : {z : Z * Z * Z | P z}) : {z : Z * Z * Z | P z} :=
  f_aux x.
Next Obligation. unfold f_aux; unfold P; inversion H; lia. Qed.
```

With this new version of `f` the exported code look as one would expect. Indeed, it exports to

```
let f_aux (x : ((i64,i64),i64)) = match x
  case (p, x3) -> (match p case (x1, x2) ->  ( (x2, x1), x3))

let f (x : sig_ ((i64,i64),i64)) = (f_aux (id x))
```

This trick with creating an auxiliary function was used when defining the associate operator for the maximum segment sum function, `mss`. That is, we defined `redOp` in terms of a function `redOp_aux` without subset types, as

```
Definition max (x y : Z) : Z := if x >? y then x else y.

Definition redOp_aux (x y : Z * Z * Z * Z) : Z * Z * Z * Z :=
  let '(mssx, misx, mcsx, tsx) := x in
  let '(mssy, misy, mcsy, tsy) := y in
  ( max mssx (max mssy (mcsx + misy))
  , max misx (tsx + misy)
  , max mcsy (mcsx + tsy)
```

[29] For the interested reader, the result of `Print f` has been included in Section B.3 in the appendix.
[30] Here `sig_` is the identity on types, that is `type sig_ 'a = a`.

```
  , tsx + tsy ).

Program Definition redOp (x y : X) : X :=
  redOp_aux (proj1_sig x) (proj1_sig y).
```

so that `redOp_aux` is defined according to (7), but on all of $\mathbb{Z}^4$, and `redOp` is the restriction of this function to the domain we are interested in.

Now, besides the choice of domain for the operator, the above problem was the main difficulty. Actually providing an instance of the `IsMonoid` class was, to a large degree, something that could be solved by the `lia` tactic (linear integer arithmetics). However, in order for `lia` to be able to tackle the problem, we needed to go through a tedious process of preparing the goal and the hypothesis. This involved unfolding and splitting the condition of the subset type and converting boolean inequalities to propositions and such. This part involved writing a lot of tactics in order to get this automated, more or less culminating in the tactic `destruct_Xs` which takes each element of type `X` and decomposes it into four integer values and proofs of the inequalities that they satisfy as elements of `X`.

With the `IsMonoid` criteria in place, the definition of the maximum segment sum function was quite simple, indeed, with the obligation solving removed, it is given as

```
Definition X__unit               : X := exist _ (0, 0, 0, 0) eq_refl.

Program Definition mapOp (x : Z) : X := (max x 0, max x 0, max x 0, x).

Definition mss_core {n : nat} (xs : [|n|]Z) : X :=
  reduce redOp X__unit (map mapOp xs).

Definition mss {n : nat} (xs : [|n|]Z)      : Z :=
  let '(x, _, _, _) := proj1_sig (mss_core xs) in x.
```

As opposed to the introduction of `redOp_aux`, the function `mss_core` is not introduced to combat extraction problems, but rather to make it easier to deal with functional correctness in Section 5.5.

Now, the extracted Futhark code from the `mss` function can be found in Section B.1 in the appendix.

5.5 Functional correctness

Functional correctness obviously means that the result of the function we have defined is indeed the maximum segment sum. In order to formulate this in Coq, we make use of the concept of a segment mentioned in 4.3. With this notion, the statement of functional correctness can be stated by the following two results:[31]

```
Theorem mss_bound {n1 n2 : nat}:
  forall (l1 : [|n1|]Z) (l2 : [|n2|]Z), Segment l1 l2 -> sum l1 <= mss l2.
```

[31] The `sum` function is, of course, the function that returns the sum of an array.

```
Theorem mss_attain:
  forall (n2 : nat) (l2 : [|n2|]Z), exists (n1 : nat) (l1 : [|n1|]Z),
    Segment l1 l2 /\ sum l1 = mss l2.
```

The former of the two states that the value of the `mss` function is an upper bound for all segment sums, and the latter states that there is indeed a segment whose sum is the value of the function. Clearly this is what functional correctness of the `mss` function means.

Now, in order to prove this, we prove a series of results about the `mss_core` function. Indeed, we prove that the two first coordinates returned by this function (partially) have the semantics described in Section 5.3. More precisely, we prove

```
Lemma mss_core_initial {n : nat}:
  forall l : [|n|]Z,
    let '(_, msil, _, _) := proj1_sig (mss_core l) in
    exists (n' : nat) (l' : [|n'|]Z), Prefix l' l /\ sum l' = msil.
```

which states that the second value is actually the sum of an initial segment, and we prove

```
Lemma mss_core_inner {n : nat}:
  forall l : [|n|]Z,
    let '(mssl, _, _, _) := proj1_sig (mss_core l) in
    exists (n' : nat) (l' : [|n'|]Z), Segment l' l /\ sum l' = mssl.
```

which states that the first value is also the value of a segment, but not necessarily initial. With these results we are able to prove the functional correctness results mentioned above.

We will not go into details with the proofs, but just mention a few of the main points. The main work is done via automation with the `auto` and `autorewrite` tactics, using the hint databases `futhark` and `mss`. Besides this, tactics are made for destructing values of the form `mss_core l` for some array `l`. Besides these methods, the most complicated part consists of splitting in cases depending how the values of `mss_core` on an array are updated based on the previous array. For example, during the proof of `mss_core_inner` we have (something similar to) the following

```
match goal with
| |- context[max (max h 0) (max ?mss' (max h 0 + ?mis'))]
  => remember (max (max h 0) (max mss' (max h 0 + mis'))) as MSS
     assert (MSS_CASES : MSS = mss' \/ MSS = h \/ MSS = h + mis') by (...)
end.
```

A careful examination of the expression we try to find in the goal will reveal that this is actually the expression in the first coordinate in (7) when the left hand coordinates correspond to the singleton list `[h]`. Thus, we see that the `MSS_CASES` is a dichotomy for how the maximum segment sum can update when `h` is added to the front of the list.

5.6 The certificate

At this point we have explained how we have implemented the `mss` function, calculating the maximum segment sum, so that it extracts to Futhark code, and how we have verified

that the function is indeed correct. However, a priory, the proofs we have made, of associativity and functional correctness say, only guarantee that the Coq code has these properties. The question is, what can we say about the extracted Futhark code, and which guarantees do we get.

First of all, as always, guarantees comes with some assumptions based on trust. In our case, there are some obvious candidates, namely, the pretty printer and the function remapping. Any guarantee for the Futhark code must necessarily be prefixed by an assumption that the pretty printer is correct and that the (possibly user supplied) remappings are sensible. We have already discussed in Section 4.2 which precautions can be taken to try and ensure that remappings of functions make sense, and for the pretty printer, we, at the moment, can find solace in the fact that the pretty printer is a rather small amount of code. We will not talk much more about these assumptions and other similar ones, and just assume for a moment that we can trust that the Futhark code is extracted faithfully from the Coq code.

Now, with these assumptions, it still leaves the more serious problem that we remap the integer type `Z` in Coq to the 64 bit integer type `i64` in Futhark. Since the former of the two has unbounded precision whereas the latter has bounded precision, the results about the extracted Futhark code do not carry over verbatim.

So what can we say about the extracted Futhark code. Well, if no overflow occurs, then the guarantees all work and the results proved in Coq carry over. Indeed, it is true for any bounded integer type that, if no overflow occurs, then the result is the same as in unbounded integer arithmetics. Now, as Futhark generates data-parallel code in intricate ways, we are not even guaranteed that the same array will always produce the same result if an overflow occurs, as the way the data is divided during parallel execution might vary. For this reason it might be practical to know whether an overflow occurred or not, and it might be worth trying to detect that. In the case you know if there has been an overflow, you know that you can trust the result in case there is none.

5.7 Similar problems

Let us just mention that there are several problems that can be solved in a similar manner to the maximum segment sum, and which might be implemented and certified using the implementations we have made, with some corrections. One of these problems consists of finding the length of the longest increasing sequence of contiguous elements in an array. It is easy to see that we can come up with an associative operator in a similar manner to the one for the maximum segment sum, by again holding information on which element an array start and end with, what the longest initial and closing sequences are, and such.

SECTION 6

Evaluation

In this section we will evaluate the example, what the pros and cons are of using the described method for extracting certified Futhark code, as well as the quality of the

implementation we have made during the project.

6.1 Quality of the implementation

During implementation, it has been a high priority to use custom automation and tactics, as well as separating out common elements in order to make proofs simple. To a large degree we feel that we have succeeded in this goal, and that very few of the proofs are long or complicated. Many of the common patterns, like destructing element of the subset type used for the restricted domain of the associative operator used in the maximum segment sum example, have been delegated to tactics that can be applied in multiple proofs.

Besides writing custom tactics and creating hint databases for use with `auto` and `autorewrite`, we have also created a number of induction and destruction principles. These likewise cover patterns often used in proofs, but they also serve the purpose of hiding the problems/annoyances that come with using subset types as the model for Futhark arrays. We feel that, with the current implementation, it should only occur in very rare cases that the user would have to destruct an element of type `[|n|]A` into the form `exist _ xs pf`. This fact seems quite essential, in order to make the FuCert framework sufficiently easy to use for a programmer unfamiliar with the details of the implementation.

Now, the automation made as part of the infrastructure is, of course, available for the next user of the framework, but we also feel like the custom tactics used in the maximum segment sum example might well work for many other cases as well, following some adaptions and additions. Indeed, much of it is mainly concerned about dealing with subset types where the underlying type is tuples of integers and the conditions are linear equations.

The maximum segment sum example is complete, but the FuCert infrastructure is not quite so, yet. Completeness of the FuCert infrastructure was not a goal of this project though, and, all in all, it is our belief that the current implementation is of good quality. However, as this is the author's first encounter with Coq (or any proof assistant), there might be certain things that are not done in the most optimal way due to lack of experience. We believe, though, that such problems would not be too difficult to correct, and that there are no fundamental problems.

6.2 Limitations and disadvantages of FuCert

In this section, we will discuss some of the limitations and disadvantages of FuCert, as well as which of these are inherent to this approach and which ones could possibly be solved further down the road. Let us start with the limitations.

We have already mentioned some of the problems one might encounter in using Coq to extract Futhark code using the FuCert framework earlier. One of the problems, described in Section 5.4, with whether the extracted code is valid Futhark code. Indeed, a definite problem is that extraction will never fail, even if the resulting code is not valid Futhark code. Indeed, in some cases, the extracted code might be sufficiently complicated so as

to make it difficult determine which part is not extracted correctly, and as we saw earlier, this problem with extraction can be difficult to foresee when using something like the `program` tactic.

Another problem that we have not touched upon yet, are places where the extraction is too simplistic and fails in generating valid Futhark code, because it fails in supplying correct type information. One way this manifests itself is in the fact that the extracted code does not contain any size parameters. This means that we run the risk of extracting to a program that is not valid unless the correct size parameters are inserted. Another way this manifests itself is in that the extraction does not insert type parameters, so that extraction of polymorphic functions does not work. For example, the function

```
Definition f {A : Type} {n : nat} (xs : [|n|]A)
  : ([|n|]A) * ([|n|]A) := (xs, xs).
```

extracts to

```
type sig_ 'a = a
let f (xs : sig_ ([] a0)) =  (xs, xs)
```

which is obviously not valid Futhark, and yields an *Unknown type "a0"* message.

The problems described so far might seem quite significant. However, it might be that many of these can be solved to a greater or lesser degree. The problem with not knowing if the extraction resulted in a valid Futhark programs can somewhat be solved by adding an intermediate step to the pipeline, so that we instead of printing the $\lambda^T_\square$ program directly, translate into an abstract syntax tree representation of (a subset of) Futhark, which we then print. In this way we check that the Futhark representation of the program is (close to) valid, before printing it.

When it comes to the problem of polymorphic types, it seems like it might be possible to modify the pretty printer so that it inserts these type parameters. However, what it comes to size types, this is certainly not that easily solved, as we have erased the relevant type information (the sizes) at the time we invoke the pretty printer. The ConCert framework has some functionality to keep some information around when performing its erasure procedure on types, so that it might be recovered, but it is not clear how this should be translated to valid Futhark, as general expressions are not allowed as type parameters.

Now, besides the mentioned limitations, there is obviously also the fact that not too much Futhark infrastructure have been implemented yet, and much remapping and exporting is still done on a case by case basis. However, these are obviously problems that can be solved by implementing the rest of this infrastructure.

Let us talk a little about the disadvantages of using this method for extracting certified Futhark code. A disadvantage of this method of extraction is obviously that writing Coq is quite far away from the Futhark domain, and for a programmer to use these methods, he or she would need to be experienced in Coq as well as Futhark. Another disadvantage of this method is that the extracted code might not be very easily readable. We have already seen examples of extracted code, and it can end up seeming obscure. This latter

fact does not seem like a big problem though, as extracted code would likely be put in a library, and all details hidden behind a Futhark module type.

6.3 Advantages of FuCert

Now that we have talked about the disadvantages of using the FuCert framework, let us talk a bit about the advantages. The main advantage is, obviously, that users have the full power of Coq at their disposal. For this reason, verification of the desired properties is, in some sense, not more complicated than the problem itself, that is, as there are not other restrictions to proving it in Coq. In particular, as all values in the `Prop` universe are stripped away during extraction, there is a lot of flexibility in having function definitions carry certificates around.

We mentioned earlier that a disadvantage of FuCert was that it would export anything, even if it is not valid Futhark. However, this also turns out to be an advantage, in the sense that there are not many limitations to what can be exported. If users had to write programs written using a particular subset of Coq or an available set of functions, it would be more restrictive and likely less suited for some verifications.

Another benefit of the FuCert framework is the large flexibility. There is some infrastructure, in terms of Futhark functions, but the user is free to use his or her own. In particular, as there are no boundaries on remapping, the user can use this to bypass limitation/problems in the export for complicated functions.

It is also worth mentioning that the automation we have implemented so far, in terms of hint databases, might be able to relieve the user for a lot of proof burden.

SECTION 7

Conclusion

Through the process of implementing the maximum segment sum example, and needed FuCert infrastructure, it seems clear that this method for extracting certified Futhark programs is certainly viable. The fact that Coq has different universes for values and proofs, and that the latter are stripped away, makes it flexible and easy to use for generating certified Futhark code. At this point, there is still much in the FuCert framework that can be improved in order to remove limitations and increase the ease for the user, but it certainly seems like this can become a quite good tool for generating certified Futhark code, possibly in terms of verified libraries that take care of some central, optimised tasks.

References

[1] Danil Annenkov. GitHub page for FuCert framework. `https://github.com/annenkov/futhark-extract`.

[2] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. Extracting functional programs from coq, in coq, 2021.

[3] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: A smart contract certification framework in coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 215–228, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Jon Bentley. Programming pearls: algorithm design techniques. *Communications of the ACM*, 27(9):865–873, 1984.

[5] G.E Blelloch. Scans as primitive parallel operations. *IEEE transactions on computers*, 38(11):1526–1538, 1989.

[6] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[7] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive array constructs in an embedded gpu kernel programming language. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*, DAMP '12, page 21–30, New York, NY, USA, 2012. Association for Computing Machinery.

[8] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark*. DIKU, 2018.

[9] Martin Elsman, Troels Henriksen, and Niels Gustav Westphal Serup. Data-parallel flattening by expansion. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2019, page 14–24, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 556–571, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The metacoq project. *Journal of automated reasoning*, 64(5):947–999, 2020.

[12] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

APPENDIX A

Source code overview

In this section, we will give an overview of the accompanying source code to this project. It consists only of Coq code, and can be found here:

`https://github.com/okknudsen/futhark-extract/tree/ba`

that is, in the `ba` of the linked Git repository. This repository is a fork of the official FuCert Git repository[32]

A.1 Running the code

A guide on how to setup the FuCert framework, with its dependencies, using `opam` can be found in the `README.md` file of the FuCert repository.

The code have been tested with *Coq version 8.11.2* and the extracted code have been tested with *version 0.19.2* of the Futhark compiler. Instructions on how to install the Futhark compiler can be found in the Installation section of the Futhark User's Guide.[33]

After setting up the FuCert framework, it can be compiled, and the example Futhark code extracted, by running `make` in the root of the FuCert project.[34]

After compilation, the test included in the beginning of some of the Futhark files can be executed with `make test-extraction`. This requires the Futhark compiler to be installed.

A.2 Included files

In this section we will give an overview of the source files in (our clone of) the FuCert repository, explaining what role each file has, as well as what our contribution to the source is.

Besides for files related to dependencies, setup, automation and continuous integration, the files are divided into two folders, `extracted` and `theories`. The first of these folders simply contains extracted Futhark code. Besides the examples existing in the FuCert framework, there are the files

`extract/mss.fut` This file contains the extraction of the maximum segment sum example (displayed in Section B.2)

`extract/mss_unsized.fut` This file contains the extracted code from the unsized implementation of the maximum segment sum. We will elaborate on what this means below.

[32] `https://github.com/annenkov/futhark-extract`

[33] `https://futhark.readthedocs.io/en/stable/installation.html`

[34] If Coq and the FuCert dependencies where installed via `opam`, make sure to execute `make` (and following instructions) in the correct `opam` environment, for example with `opam exec make`.

Now, the `theories` folder is where the actual implementation is placed. There are a few files which were already part of the FuCert framework at the beginning of this

`theories/FutharkExamples.v` This contains the pretty printer from $\lambda^T_\square$(of ConCert) to Futhark.

`theories/FutharkPretty.v` This contains wrapper functions for the ConCert extraction functionality and file extraction.

`theories/FutharkExtract.v` This contains extraction examples displaying the FuCert frameworks extraction capabilities.

Besides these files, there are a bunch of files of our contribution. These fall into three categories (with a bit of overlap), namely, files related to FuCert infrastructure, as described in Section 4.4, fiels related to the maximum segment sum example, as described in Section 5, and, lastly, an implementation of the maximum segment sum example that does not use the arrays described 4.3, but rather plain lists. Of the files related to Futhark infrastructure, there are:

`theories/FutharkUtils.v` This file contains shared code having to do with Futhark infrastructure, such as a type class describing the monoid criteria. Many of thse are described in Section 3.2

`theories/FutharkArrays.v` This file contains the functionality surrounding the chosen array model, as described in Section 4.3.

`theories/FutharkMod.v` This file contains the module type and functor described in Section 4.2 in order to specify the Coq variant of the Futhark functions, as well as extra results about these.

`theories/FutharkModImpl.v` This file contains the actual implementation of the specification of the Coq version of the Futhark function and a justification that they satisfy the required axioms, as described in Section 4.4

`theories/FutharkUnsized.v` This file contains the analogue of the specification and implementations of Futhark functions when using lists instead of the array model described in Section 4.3.

Of the files that are concerned with the maximum segment sum example (both of them)

`theories/MssUtils.v` This file contains code that is relevant to both implementation of the maximum segment sum example, as described in Section 5.4. This turns out to be almost all of it.

`theories/MssSizedDefinition.v` This (very short) file contains the definition of the maximum segment sum function for the array model we have described in this project.

`theories/MssSizedCorrectness.v` This function contains proof of functional correctness of the maximum segment sum function, for the array model we have chosen, as described in Section 5.5.

`theories/MssUnsizedDefinition.v` This (very short) file contains the definition of the maximum segment sum function for the version with lists.

`theories/MssUnsizedCorrectness.v` This file contains the proof of functional correctness of the maximum segment sum function for the version that uses lists.

`theories/MssExtract.v` This file contains code for extracting the two versions of the maximum segment sum functions, using the ConCert framework and the FuCert pretty printer.

APPENDIX B

Various code listing

This section contains various code listings that we have chosen not to include, but that the reader might want to take a look at out of interest.

B.1 Maximum segment sum in Futhark

The following is the Futhark implementation of the maximum segment sum; taken from the futhark-lang.org.[35]

```
1  -- Parallel maximum segment sum
2  -- ==
3  -- input { [1, -2, 3, 4, -1, 5, -6, 1] }
4  -- output { 11 }
5
6  let max(x: i32, y: i32): i32 =
7    if x > y then x else y
8
9  let redOp(x: (i32,i32,i32,i32))
10          (y: (i32,i32,i32,i32)): (i32,i32,i32,i32) =
11   let (mssx, misx, mcsx, tsx) = x in
12   let (mssy, misy, mcsy, tsy) = y in
13   ( max(mssx, max(mssy, mcsx + misy))
14   , max(misx, tsx+misy)
15   , max(mcsy, mcsx+tsy)
16   , tsx + tsy)
17
18 let mapOp (x: i32): (i32,i32,i32,i32) =
19   ( max(x,0)
20   , max(x,0)
```

[35] Direct link the file: `https://futhark-lang.org/benchmarks/programs/mss.fut`.

```
  , max(x,0)
  , x)

let main(xs: []i32): i32 =
  let (x, _, _, _) =
    reduce redOp (0,0,0,0) (map mapOp xs) in
  x
```

B.2 Extracted version of mss

Here we show the extracted version of the `mss` function, calculating the maximum segments sum, which we described the implementation and verification of in Section 5

```
-- Maximum segment sum test
-- ==
-- input { [1i64,-2i64,3i64,4i64,-1i64,5i64,-6i64,1i64] }
-- output { 11i64 }

type sig_ 'a = a
let Exist = id
let addI64 (i : i64) (j : i64) = i + j
let subI64 (i : i64) (j : i64) = i - j
let multI64 (i : i64) (j : i64) = i * j
let divI64 (i : i64) (j : i64) = i / j
let leI64 (i : i64) (j : i64) = i <= j
let ltI64 (i : i64) (j : i64) = i < j
let geI64 (i : i64) (j : i64) = i >= j
let gtI64 (i : i64) (j : i64) = i > j
let eqI64 (i : i64) (j : i64) = i == j
let map_wrapper    [n] 'a 'x (m: i64) (f: a -> x) (as: [n]a) : *[n]x        =
↪  map f as
let reduce_wrapper [n] 'a    (op: a -> a -> a) (ne: a) (m: i64) (as: [n]a) : a
↪  = reduce op ne as
let scan_wrapper   [n] 'a    (op: a -> a -> a) (ne: a) (m: i64) (as: [n]a) :
↪  *[n]a = scan op ne as
let zip_wrapper    [n] 'a 'b (m: i64) (as: [n]a) (bs: [n]b)  : [n](a, b)    =
↪  zip as bs
let unzip_wrapper  [n] 'a 'b (m: i64) (xs: [n](a, b))        : ([n]a, [n]b) =
↪  unzip xs


type x = sig_ (((i64,i64),i64),i64)

let max (x : i64) (y : i64) = if gtI64 x y then x else y

let redOp_aux (x : (((i64,i64),i64),i64)) (y : (((i64,i64),i64),i64)) = match x
case (p, tsx) -> (match p
case (p0, mcsx) -> (match p0
case (mssx, misx) -> (match y
```

```
case (p1, tsy) -> (match p1
case (p00, mcsy) -> (match p00
case (mssy, misy) ->  ( ( ((max mssx (max mssy (addI64 mcsx misy))), (max misx
↪  (addI64 tsx misy))), (max mcsy (addI64 mcsx tsy))), (addI64 tsx tsy)))))))

let redOp (x : x) (y : x) = (redOp_aux (id x) (id y))

let X__unit  =  ( ( (0i64, 0i64), 0i64), 0i64)

let mapOp (x : i64) =  ( ( ((max x 0i64), (max x 0i64)), (max x 0i64)), x)

let mss_core (n : i64) (xs : sig_ ([] i64)) = reduce_wrapper redOp X__unit n
↪  (map_wrapper n mapOp xs)

let mss (n : i64) (xs : sig_ ([] i64)) = match id (mss_core n xs)
case (p, z) -> (match p
case (p0, z0) -> (match p0
case (x, z1) -> x))

let mss_wrapper (xs : [] i64) = mss (length xs) (id xs)

let main = mss_wrapper
```

B.3 Example function defined with Program tactics

This is the definition of the example function `f` in Section 5.4, showing that seemingly simple functions defined with the program tactics can result in a complicated definition. The definition of the function, gotten with `Print f`, is

```
f =
fun x : {z : (Z × Z) × Z | P z} =>
let filtered_var := ` x in
let program_branch_0 :=
  fun (x1 x2 x3 : Z) (Heq_x : (x1, x2, x3) = filtered_var) =>
  exist (fun z : (Z × Z) × Z => P z) (x2, x1, x3)
    ((fun x0 : {z : (Z × Z) × Z | P z} =>
      let filtered_var0 := ` x0 in
      fun (x4 x5 x6 : Z) (Heq_x0 : (x4, x5, x6) = filtered_var0) =>
      f_obligation_1 x0 x4 x5 x6 Heq_x0) x x1 x2 x3 Heq_x) in
(let
 '(p, x3) as x' := filtered_var return (x' = filtered_var -> {z : (Z × Z) × Z |
↪  P z}) in
  let
  '(x1, x2) as p0 := p return ((p0, x3) = filtered_var -> {z : (Z × Z) × Z | P
↪  z}) in
   program_branch_0 x1 x2 x3) eq_refl
     : {z : (Z × Z) × Z | P z} -> {z : (Z × Z) × Z | P z}
```

Though it looks complicated, it is even more complicated, as Coq did not expand the functions `f_obligation_1` and `program_branch_0`, generated by the `program` tactic.