FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

BSc Project

Christian Kjær Larsen

Formalization of Array Combinators and their Fusion Rules in Coq

An experience report

Supervisor: Martin Elsman

June 12, 2017

Abstract

There has recently been new large developments in the field of compiler correctness. Projects like CompCert and Vellvm try to build verified software for compilers that are proven correct with respect to its specification. There has also been new research on efficient implementations of data-parallel array programming languages on GPU's. We present a framework for reasoning about array programs and transformations with respect to formal semantics using the Coq proof assistant. We attempt some proofs of correctness of various program transformations that one would do in a realistic compiler, and we investigate the possibility of extracting programs for transforming array-programs that are proven correct and then can be embedded in a real world compiler.

Contents

1. Introduction

In this section we will give a quick introduction to the domain of our work, and we will explain the goals and the motivation of the work.

1.1 Compiler correctness

The field of compiler correctness deals with showing that a compiler behaves according to its specification. Compilers like GCC or MSVC typically has large validation suites that that checks that the compiler behaves appropriately in a large number of test cases.

Another way of verifying that a compiler behaves according to its specification is using formal methods. If there are formal semantics for the programming languages involved, then one can prove that every step of the compilation process preserves the program semantics.

Source code transformations and optimizations should also preserve semantics. Simple examples of transformations includes in-lining, constant folding and common sub-expression elimination.

1.2 Motivation

The main motivation for this project is to investigate methods for verifying correctness of program transformations. Many compilers do optimizations to make compiled programs perform better, and some of these optimizations can introduce bugs in the compiled programs.

A major effort on compiler correctness has been the CompCert [1] C compiler. It is a verified compiler for a large subset of the C programming language written in the Coq proof assistant. Being verified means that along side the compiler is a machine checked proof of its correctness with respect to the language semantics. CompCert also does optimizations that are verified as well.

The compiler can then be extracted from the proof assistant and compiled using the OCaml programming language.

Another large project is CakeML [2], which is a mechanically verified implementation of a substantial subset of Standard ML developed using the HOL4 theorem prover. It has been proved that the CakeML compiler transforms CakeML programs into semantically equivalent machine code.

The long-term goal of this work is to create a verified fusion engine for programs written using a data-parallel programming language with various array combinators. This fusion engine would then be proven correct, extracted and embedded into an existing compiler.

1.3 Contribution

Our contribution is the investigation of a possible framework for verifying program transformations for a data-parallel programming language with second-order array combinators using the proof assistant Coq. In particular we will investigate fusion rules for map and filter and verification of their correctness. Furthermore we see if more complicated encoding techniques makes proofs easy to formalize.

We have not been able to find existing literature about this topic, so we find the project quite innovative and new.

We will start by giving some background on second-order array combinators.

2. Second-order array combinators

Second-order array combinators are limited forms of some of the higher order functions on lists/arrays found in languages like Haskell and SML. They are limited in the sense that they do not allow function objects as arguments, but they only allow for syntactic anonymous functions. We will mainly focus on map and filter in this work.

$$\texttt{map} : \forall\alpha\beta.(\alpha \to \beta) \to [\alpha] \to [\beta] \tag{2.1}$$

$$\texttt{filter} : \forall\alpha.(\alpha \to \texttt{bool}) \to [\alpha] \to [\alpha] \tag{2.2}$$

In the case of `map`, we can define a second order version of this function by fixing the bound variable, and only allow invocations in the form

$$\texttt{map}\ (\lambda x.f)\ e \tag{2.3}$$

Where the intuition is, that x is bound to each element of e in the evaluation of f, the body of the function.

2.1 APL

A language that pioneered this way of programming was the language APL developed in the 60s by Kenneth Iverson [3]. It has multidimensional arrays as its central data type. APL defines a number of built in operators on arrays. They are used to specify the way that functions are applied on arrays. For instance one can map the function that increments by 2 onto an array by doing

```
  2 + ¨ 1 2 3 4
3 4 5 6
```

Or the reduction with addition can be performed by doing

```
  +/ 1 2 3 4
10
```

APL also defines operators for scan, filter and more. Languages in this style were typically faster than regular interpreted programming languages, because APL style languages have a lower interpretation overhead, and they can leverage highly optimized implementations of these operators perhaps even using parallelism.

APL is a dynamically typed language, and there has been many attempts at compiling APL down to efficient machine code. Budd [4] noticed that a lot of APL operators does not alter values, but merely changes representations. This observation is used to construct an efficient APL compiler. Budd also talks about the concept of *drag-through*, which is when an operation is pushed through other operators.

Elsman and Dybdal [5] has demonstrated how many of these constructs can be efficiently compiled down to a typed lower level representation.

This work includes giving explicit types for some of the operators in APL. One that is relevant for this project is the each operator

$$\texttt{each} : \forall abc.(a \to b) \to [a]^c \to [b]^c \tag{2.4}$$

This definition is almost the map function, but the difference is that it works on multidimensional arrays as well. This property can be seen by the universally quantified shape parameter c.

2.2 Futhark

One of the primary reasons for doing the project is the programming language Futhark, that is currently under development at the University of Copenhagen [6]. Futhark is a purely functional array language and a compiler that generates highly optimized code for GPUs.

The basis for Futhark is a number of second-order array combinators. Performance is achieved from semantics-preserving transformations that that optimizes for temporal and spacial locality of reference.

The correctness arguments for these transformations come from the list-homomorpishm theory of Bird and Meertens [7]. This project will take another approach, and try to investigate some of these transformations in a less abstract setting. We will look into proving correctness of transformation rules using formal semantics for a small programming language.

2.3 Fusion rules

A common compiler optimization that we want to do in a programming language with these array combinators is fusion. One example of an optimization is fusing two nested maps into one. This optimization removed one array traversal. In a programming language with higher-order functions, defining a fusion rule for nested maps is pretty easy

$$\texttt{map}\, f\, (\texttt{map}\, g\, e) \equiv \texttt{map}\, (f \circ g)\, e. \tag{2.5}$$

The problem with this example is, that we only allow expressions written like formula 2.3, because the previous definition needs higher order functions. To resolve this problem we need to define the previous fusion rule as a syntactic transformation using substitution. One way of doing this transformation is by substituting in the function body

$$\texttt{map}\, (\lambda x.f)\, (\texttt{map}\, (\lambda x.g)\, e) \equiv \texttt{map}\, (\lambda x.[x := g]f)\, e. \tag{2.6}$$

One potential problem here is, that we can end up repeating computation. If x appears multiple times in f, then we repeat the evaluation of g. If the compiler for our language implements common sub-expression elimination, then we can safely assume that this optimization does not introduce additional computational overhead.

We can also eliminate the problem by introducing a let binding.

$$\texttt{map}\, (\lambda x.f)\, (\texttt{map}\, (\lambda x.g)\, e) \equiv \texttt{map}\, (\lambda x.\texttt{let}\ x' \Leftarrow g\ \texttt{in}\ [x := x']f)\, e \tag{2.7}$$

where x' is fresh. Then we do not introduce more computation, but we bind an additional variable.

Another fusion rule that one can have is combining filters

$$\texttt{filter}\, p\, (\texttt{filter}\, q\, e) \equiv \texttt{filter}\, (\lambda x.p\, x \wedge q\, x)\, e. \tag{2.8}$$

This rule is simpler, as it does not depend on doing substitution, we can just insert the two lambda-bodies into a new one, where they are combined using the $\wedge$ operator.

There are many other fusion rules than these two, but we will mainly focus on map and filter fusion in our work. Other fusion rules are described in section 7.

Another way to think of these combinators is using category theory. We will make a quick aside on the categorical properties of some of these combinators.

2.4 A categorical treatment

This section will give a very brief introduction to how category theory can be used to reason about functions on lists. We will skip the most abstract concepts, and skip directly to the application to a functional programming language. This simplification means that morphisms are functions, and objects are types. This category is the actual category called `Hask`, that is used in the Haskell programming language. Some say that Haskell is a direct implementation of category theory.

A functor is defined as a mapping between categories [8]. More concretely for two categories $\mathbf{A}, \mathbf{B}$, a functor between them is written as $\mathbf{F} : \mathbf{A} \to \mathbf{B}$.

A functor consists of two mappings, one takes functions to other functions, and one takes types to other types. The mapping that takes types to other types is called the type constructor. The mapping that takes functions to functions is in Haskell called `fmap`. It has the type signature

```
fmap :: (a -> b) -> F a -> F b
```

There are two laws that functors must obey.

1. $\mathbf{F}(\mathrm{id}_A) = \mathrm{id}_{\mathbf{F}(A)}$
2. $\mathbf{F}(g \circ f) = \mathbf{F}(g) \circ \mathbf{F}(f)$

We can state these laws is terms of Haskell code as well. The first law then states that

```
fmap id = id
```

and the second law states that

```
fmap (g . f) = (fmap g) . (fmap g)
```

A lot of data data types in Haskell are functors, and we can define a list data type in Haskell in the following way.

```
data List a = Nil | Cons a (List a)
```

And the canonical map function is implemented as follows.

```
map :: (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

Now if `List` is a functor, then we know directly from the second functor law that map fusion is correct. We can prove these functor laws for our list type by induction.

Proof of functor laws

A common proof strategy in Haskell is one of equational reasoning. We can expand function definitions and do beta-reductions. We can do this style of reasoning because Haskell is pure and referentially transparent. Then we can replace equal terms with each other.

We will prove the second functor law for `map`. We do the proof by structural induction over the `List` data type.

Our inductive hypothesis is that for all lists `xs`,

```
map (g . f) xs  = ((map g) . (map f)) xs
```

We start with the base case.

```
map (g . f) Nil = map g (map f Nil)
-- unfold map
Nil = map g Nil = Nil
```

For the inductive case we assume that

```
map (g . f) xs = map g (map f xs)
```

And we need to prove that

```
map (g . f) (Cons x xs) = map g (map f (Cons x xs))
-- unfold map
Cons ((g . f) x) (map (g . f) xs) = map g (map f (Cons x xs))
-- Use induction hypothesis
Cons ((g . f) x) (map g (map f xs)) = map g (map f (Cons x xs))
-- unfold map
Cons ((g . f) x) (map g (map f xs)) = Cons (g (f x)) (map g (map f xs))
-- function composition
Cons (g (f x)) (map g (map f xs)) = Cons (g (f x)) (map g (map f xs))
```

And we are done.

This proof strategy is very common when working with data types in Haskell and other functional programmings languages. If some data type satisfies the laws for functors, applicatives or monads, then we get some properties handed to us.

In the remainder of this report we will explore some possible ways of verifying the fusion rules for map and filter. We start by defining a language to base our work on.

3. Introducing `ling` (LIst laNGuage)

To be able to formalize array combinators and their corresponding fusion rules, we need to have a language for doing so. It is not feasible to use an already existing programming language like APL, Futhark or Haskell. They are much too large for our purpose, and this makes reasoning about the meaning of various programs very tedious.

Therefore we have chosen to create a small language with only the features we need for proving properties about array combinators.

3.1 Design

The language is a small pure functional programming language with call by value semantics. It has no recursion or functions, and all computation must be expressed in terms of built in combinators. We do not formalize arrays, but we use lists instead. They are much easier to reason about, and they are isomorphic to arrays.

This choice means that some of the constructs in the language will not be accessible to the user, and are only here for reasoning purposes. An array language will for instance not have `nil` and `cons` available. The language will in essence only have the array combinators available for programming. Furthermore, to make the programming language useful, it will also need more general language constructs to make real programs expressible. We have not included these constructs in this work, because they will make the reasoning process much more complicated.

Many formalizations of programming languages treat values as a subset of the syntax, and then define a relation for what it means to be a value. We have chosen a separate syntax for values, and they are then different objects. This decision also restricts us to creating only a big-step operational semantics. This choice is made because this evaluation strategy models more closely the way that a language like this is implemented in practice.

The language includes natural numbers, booleans, and lists of those values. The language features includes let-bindings, map, filter, and append.

We will in the rest of this chapter go through the complete syntax, semantics, and typing rules for the language.

3.2 Syntax

We give the syntax for expressions and values. As stated before, values and expressions are different syntactic categories.

$$
\begin{array}{l}
e ::= x \\
\quad \mid \texttt{let}\ x \Leftarrow e_1\ \texttt{in}\ e_2 \\
\quad \mid \overline{n} \\
\quad \mid \texttt{true} \mid \texttt{false} \\
\quad \mid \texttt{S}\, e \\
\quad \mid e_1\ \texttt{and}\ e_2 \\
\quad \mid \texttt{nil} \mid e_1 :: e_2 \\
\quad \mid \texttt{map}\ (\lambda x.e_1)\ e_2 \\
\quad \mid \texttt{filter}\ (\lambda x.e_1)\ e_2 \\
\quad \mid e_1 +\!\!+ e_2
\end{array}
\qquad
\begin{array}{l}
v ::= \overline{n} \mid \texttt{true} \mid \texttt{false} \\
\quad \mid \texttt{nil} \mid v_1 :: v_2
\end{array}
$$

3.3 Typing

As described before, we give the syntax for types. We have natural numbers, booleans and lists in our language.

$$\tau ::= \texttt{nat} \mid \texttt{bool} \mid [\tau]$$

We have a typing judgment for expressions, and we say that an expression has type τ in some typing environment Γ. We also give the syntax for this typing environment

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

We can now give the typing rules for our language. They are what you would expect. In the cases where we have bindings, we expand the typing environment with the bound variable.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}(\text{T-Var})$$

$$\frac{\Gamma \vdash e_1 : \tau' \quad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x \Leftarrow e_1\ \texttt{in}\ e_2 : \tau}(x \notin \text{dom}\ \Gamma)(\text{T-Let})$$

$$\frac{}{\Gamma \vdash \overline{n} : \texttt{nat}}(\text{T-N}) \quad \frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{S}\, e : \texttt{nat}}(\text{T-S})$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}(\text{T-True}) \quad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}(\text{T-False})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1\ \texttt{and}\ e_2 : \texttt{bool}}(\text{T-And})$$

$$\frac{}{\Gamma \vdash \texttt{nil} : [\tau]}(\text{T-Nil}) \quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : [\tau]}{\Gamma \vdash e_1 :: e_2 : [\tau]}(\text{T-Cons})$$

$$\frac{\Gamma \vdash e_1 : [\tau] \quad \Gamma \vdash e_2 : [\tau]}{\Gamma \vdash e_1 +\!\!+ e_2 : [\tau]}(\text{T-Append})$$

$$\frac{\Gamma, x : \tau_1 \vdash e_1 : \tau_2 \quad \Gamma \vdash e_2 : [\tau_1]}{\Gamma \vdash \texttt{map}\ (\lambda x.e_1)\ e_2 : [\tau_2]}(x \notin \text{dom}\ \Gamma)(\text{T-Map})$$

$$\frac{\Gamma, x : \tau \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : [\tau]}{\Gamma \vdash \texttt{map}\ (\lambda x.e_1)\ e_2 : [\tau]}(x \notin \text{dom}\ \Gamma)(\text{T-FILTER})$$

We include typing rules for values as well. They are useful when talking about type preservation of evaluation rules.

$\boxed{v : \tau}$

$$\frac{}{\overline{n} : \texttt{nat}}(\text{TV-N})$$

$$\frac{}{\texttt{true} : \texttt{bool}}(\text{TV-TRUE}) \quad \frac{}{\texttt{false} : \texttt{bool}}(\text{TV-FALSE})$$

$$\frac{}{\texttt{nil} : [\tau]}(\text{TV-NIL}) \quad \frac{v_1 : \tau \quad v_2 : [\tau]}{v_1 :: v_2 : [\tau]}(\text{TV-CONS})$$

3.4 Semantics

We give a big-step semantics for our language. The semantics takes expressions to values. For our combinators, we have chosen to formalize them as separate judgments taking values to values. This models in some sense builtin functions in a programming language, where the arguments are evaluated beforehand.

We say that an expression e evaluates to a value v in an evaluation context ρ. This evaluation context is not unlike a variable table in a compiler. We also give the syntax for this evaluation context.

$$\rho ::= \cdot \mid \rho, x \downarrow v$$

We now give the complete semantics for our language. The interesting cases are those for append, map, and filter. They refer to an auxiliary judgment using $\Downarrow$. This judgment is a calculation judgment that calculates the result of append, map, and filter.

$\boxed{\rho \vdash e \downarrow v}$

$$\frac{(x \downarrow v) \in \rho}{\rho \vdash x \downarrow v}(\text{EV-VAR})$$

$$\frac{\rho \vdash e_1 \downarrow v' \quad \rho, x \downarrow v' \vdash e_2 \downarrow v}{\rho \vdash \texttt{let}\ x \Leftarrow e_1\ \texttt{in}\ e_2 \downarrow v}(x \notin \text{dom}\ \rho)(\text{EV-LET})$$

$$\frac{}{\rho \vdash \overline{n} \downarrow \overline{n}}(\text{EV-N}) \quad \frac{\rho \vdash e \downarrow v}{\rho \vdash \texttt{S}\ e \downarrow \overline{n+1}}(\text{EV-S})$$

$$\frac{}{\rho \vdash \texttt{true} \downarrow \texttt{true}}(\text{EV-TRUE}) \quad \frac{}{\rho \vdash \texttt{false} \downarrow \texttt{false}}(\text{EV-FALSE})$$

$$\frac{\rho \vdash e_1 \downarrow \texttt{false}}{\rho \vdash e_1\ \texttt{and}\ e_2 \downarrow \texttt{false}}(\text{EV-ANDF}) \quad \frac{\rho \vdash e_1 \downarrow \texttt{true} \quad \rho \vdash e_2 \downarrow v}{\rho \vdash e_1\ \texttt{and}\ e_2 \downarrow v}(\text{EV-ANDT})$$

$$\frac{}{\rho \vdash \texttt{nil} \downarrow \texttt{nil}}(\text{EV-NIL}) \quad \frac{\rho \vdash e_1 \downarrow v_1 \quad \rho \vdash e_2 \downarrow v_2}{\rho \vdash e_1 :: e_2 \downarrow v_1 :: v_2}(\text{EV-CONS})$$

$$\frac{\rho \vdash e_1 \downarrow v_1 \quad \rho \vdash e_2 \downarrow v_2 \quad v_1 +\!\!+ v_2 \Downarrow v}{\rho \vdash e_1 +\!\!+ e_2 \downarrow v}(\text{EV-APPEND})$$

$$\frac{\rho \vdash e_2 \downarrow v \quad \rho \vdash \texttt{map}\ (\lambda x.e_1)\ v \Downarrow v'}{\rho \vdash \texttt{map}\ (\lambda x.e_1)\ e_2 \downarrow v'}(x \notin \text{dom}\ \rho)(\textsc{Ev-Map})$$

$$\frac{\rho \vdash e_2 \downarrow v \quad \rho \vdash \texttt{filter}\ (\lambda x.e_1)\ v \Downarrow v'}{\rho \vdash \texttt{filter}\ (\lambda x.e_1)\ e_2 \downarrow v'}(x \notin \text{dom}\ \rho)(\textsc{Ev-Filter})$$

The append semantics is not unlike how you would implement it in a functional style. Notice that it takes values to values, and does not depend on anything else.

$\boxed{v_1 +\!\!+ v_2 \Downarrow v}$

$$\frac{}{\texttt{nil} +\!\!+ v \Downarrow v}(\textsc{App-Nil}) \quad \frac{v_2 +\!\!+ v_3 \Downarrow v}{(v_1 :: v_2) +\!\!+ v_3 \Downarrow v_1 :: v}(\textsc{App-Cons})$$

The semantics for map and filter are a bit more complicated. Notice that they depend on the general evaluation of expression. This means that we have to carry around the evaluation context, and we actually extend it before evaluating the functional argument. Again the implementation is almost as one would do it in a functional programming language.

$\boxed{\rho \vdash \texttt{map}\ (\lambda x.e)\ v \Downarrow v'}$

$$\frac{}{\rho \vdash \texttt{map}\ (\lambda x.e)\ \texttt{nil} \Downarrow \texttt{nil}}(\textsc{Map-Nil})$$

$$\frac{\rho, x \downarrow v_1 \vdash e \downarrow v_1' \quad \rho \vdash \texttt{map}\ (\lambda x.e)\ v_2 \Downarrow v_2'}{\rho \vdash \texttt{map}\ (\lambda x.e)\ (v_1 :: v_2) \Downarrow v_1' :: v_2'}(\textsc{Map-Cons})$$

$\boxed{\rho \vdash \texttt{filter}\ (\lambda x.e)\ v \Downarrow v'}$

$$\frac{}{\rho \vdash \texttt{filter}\ (\lambda x.e)\ \texttt{nil} \Downarrow \texttt{nil}}(\textsc{Filter-Nil})$$

$$\frac{\rho, x \downarrow v_1 \vdash e \downarrow \texttt{true} \quad \rho \vdash \texttt{filter}\ (\lambda x.e)\ v_2 \Downarrow v_2'}{\rho \vdash \texttt{filter}\ (\lambda x.e)\ (v_1 :: v_2) \Downarrow v_1' :: v_2'}(\textsc{Filter-Cons-True})$$

$$\frac{\rho, x \downarrow v_1 \vdash e \downarrow \texttt{false} \quad \rho \vdash \texttt{filter}\ (\lambda x.e)\ v_2 \Downarrow v_2'}{\rho \vdash \texttt{filter}\ (\lambda x.e)\ (v_1 :: v_2) \Downarrow v_2'}(\textsc{Filter-Cons-False})$$

3.5 Need for mutually inductive semantics

The reason for choosing a separate semantics for append, map, and filter can seem a bit random, and we will give a quick justification for it. If we imagine that we try to write evaluation rules without a separate judgment.

$$\frac{e +\!\!+ \texttt{nil} \quad e_2 \downarrow v}{e_1 +\!\!+ e_2 \downarrow v}(\textsc{App-Nil}') \quad \frac{e_1 \downarrow v_1 :: v_2 \quad e_2 \downarrow v_3 \quad v_2 +\!\!+ v_3 \downarrow v}{e_1 +\!\!+ e_2 \downarrow v_1 :: v}(\textsc{App-Cons}')$$

The case for App-Cons' is troublesome. $v_2 +\!\!+ v_3 \downarrow v$ is not a valid premise. Remember that our judgment has the form $\rho \vdash e \downarrow v$, and $v_2 +\!\!+ v_3$ is not a valid expression. This can be solved by creating a translation from values back to expressions, but we think it will be very complicated. The consequence for map and filter is that the rules will be mutually inductive, such that map and filter will refer back to the general evaluation relation.

This mutual relationship between judgments is something that we need to tackle when formalizing the semantics. It turns out to be a bit complicated. We will not do any proofs on paper, but we will state how the theorems would be stated on paper.

4. Programming in Coq

As mentioned before, the practical part of this project uses the Coq proof assistant. In this section we will go a bit into detail about how Coq works, and how it will be used in this project.

4.1 Propositions as types

Before diving into Coq, we need a small aside on the Curry-Howard correspondence [9].

In 1934, the mathematician and logician Haskell Curry observed a relationship between implications and functions. Later in 1969, William Howard pointed out that there was a similar relationship between the simply-typed lambda calculus and natural deduction. He noticed that simplifications of proofs corresponded to evaluations of programs. Furthermore, the usual introduction and elimination rules for logical connectives corresponded to constructors and destructors for different types like pairs, sums and lambda abstractions.

En essence this connection means that programs correspond to proofs and types correspond to propositions. This fact is used in Coq, such that if a theorem is defined as a type, and a proof is given as a program. Then checking that proof corresponds to type checking that program.

The first system to use this fact was the Automath project initiated by De Bruijn in 1967 [10]. It relied on the idea to treat proofs as first class objects. The original Automath system had a small kernel, and it could only do proof checking, there was no notion of interactive proof development.

4.2 About Coq

Coq is an environment for interactive development of programs and proofs. Coq is a proof assistant, meaning that is automates routine tasks like checking that proofs are valid, but it still leaves non-trivial details to the programmer.

Coq is based on a type theory called the Calculus of Inductive Constructions (CIC). CIC is based on the Calculus of constructions (CoC) which is a higher-order typed lambda calculus. CoC is strongly normalizing, which means that every term eventually reduces to a normal form. In practice this property guarantees that every program we write terminates. CIC expands on CoC by adding inductive definitions, which is very useful when proving properties about programming languages.

That CoC has a higher-order type system means that it is possible to have types that depend on values. This type system is also called a dependent type system. The canonical example is, that one can have a list type that depends on the length of the list. For instance one can have a type of lists of integers with the length 42.

CoC can be seen as an extension to the Curry-Howard correspondence discussed in the previous section. In this case we extend to correspondence to be that proofs in the full intuitionistic predicate logic corresponds to terms in the Calculus of Constructions.

The actual language in which one writes programs is called Gallina, and because it is based on CIC, all programs are terminating. When doing proofs in Coq, one typically does not explicitly write Gallina terms, but higher-level tactics written in a language called Ltac. Ltac gives the possibility of automating proofs by writing custom decision procedures. These procedures do not need to be checked, since they generate Gallina terms that are checked by Coq. On

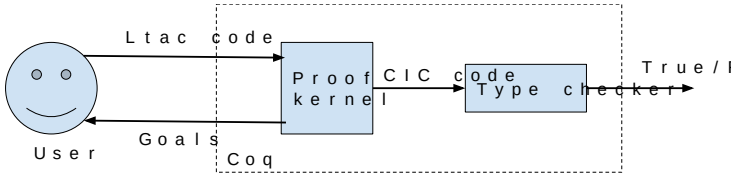

Figure 4.1: Interaction with the Coq system

figure 4.1 one can see the typical way that one would interact with the Coq system. The user is presented with a list of goals to prove. The user then interactively develops the proof by writing scripts that use Ltac tactics. Under the hood, Coq generates Gallina terms that are then type checked. If the proof is accepted, then the theorem is accepted as well.

One typically writes programs and definitions using the Gallina language, but proofs are typically written using Ltac scripts. Sometimes the converse is also true, and it often blurs the distinction between a program and a proof. We demonstrate this duality when formalizing our semantics.

4.3 Dependently typed programming

The main difference between a normal statically typed functional programming language like Haskell or ML and a dependently typed programming language like Coq, is that we can index types by values.

This feature means that we can create a list type, that is indexed by the length of the list. Then we can create for instance a function that expects lists of the same length, and have this property statically checked. This concept goes even further, and we will use dependent types to ensure that all abstract syntax trees are well typed by indexing type constructors with both types and typing environments.

In the remainder of this section we will give a short introduction to dependently typed programming by implementing a heterogeneous list type, that we will use later in this project. This implementation is partly based on examples from Adam Chlipalas book on Coq [11]. We will use this implementation later when defining evaluation contexts, because our language has different types of values that need to be stored in an evaluation context.

A heterogeneous list type

As mentioned in the previous section, a dependent type is a type that depends on a certain value. In a programming language like ML, list are typically restricted to have elements of the same type. With a dependent type system, we can create a list type that is indexed by a list of types. Then a member of the list must have the type in the list that it is indexed by.

We start by declaring a new section in Coq. A section is a way to declare local variables to a set of definitions

```
Section hlist.
```

We then declare a variable for the types in the list of types, and a wrapper for the actual value that we want to put in the list.

```
Variable A : Type.
Variable B : A → Type.
```

Just like a regular list, we have two cases. One for nil and one for cons.

```
Inductive hlist : list A → Type :=
| HNil : hlist nil
| HCons : ∀ (x : A) (ls : list A), B x → hlist ls → hlist (x :: ls).
```

The case for `HNil` is simple. We can create a heterogeneous of zero elements that is indexed by an empty list of types. The case for `HCons` is a bit more involved. Here we extend the lists that we index the type by by receiving the old list and a new element of type `x`.

An element of the list has type `A`.

```
Variable elm : A.
```

Then we can define a relation that states that some element is a member of a list. This definition is used for later to retrieve that element from the list.

```
Inductive member : list A → Type :=
| HFirst : ∀ ls, member (elm :: ls)
| HNext : ∀ x ls, member ls → member (x :: ls).
```

Now to use this list we define a function that given a proof of membership returns that particular element from the list. This function is pretty complicated, but we will go through it in detail. First we will explain a bit on dependently typed pattern matching in Coq.

Dependently typed pattern matching

Coq has an extended form of pattern matching that is very useful when doing dependently typed programming. It has the form

```
match E as y in (T x1 ... xn) return U with
| C z1 ... zm ⇒ B
| ...
end.
```

Here `E` is the expression that is matched on. It must be in the inductive type `T`. The overall type of the `match` expression is `U`. The interesting rule is, that `y` and `x1 ... xn` are bound in U. These bindings means that we can refer to the arguments of the type constructor when declaring the return type. This

property turns out to be very useful when working with dependent types. We return to the `hlist` example.

We define `hget` as a recursive function

```
Fixpoint hget ls (mls : hlist ls) : member ls → B elm :=
  match mls with
```

We delay the member argument to `hget`, then we can use the argument in a `return` clause in a pattern match. This binding is important, because we use this member argument to deal with contradictory cases.

We start with the `HNil` case

```
  | HNil ⇒ fun mem ⇒ match mem in member ls'
                       return (match ls' with
                               | nil ⇒ B elm
                               | _ :: _ ⇒ unit
                               end) with
                     | HFirst _ ⇒ tt
                     | HNext _ _ _ ⇒ tt
                     end
```

This case is contradictory, since there is no way to construct a proof of membership for the empty list. We specify this contradiction in the `return` clause. When `nil` was used to construct the member proof, then we can return the correct type, since there is no case for it.

We continue with the `HCons` case

```
  | HCons _ _ x mls'
    ⇒ fun mem ⇒ match mem in member ls'
                  return (match ls' with
                          | nil ⇒ Empty_set
                          | x'::ls'' ⇒ B x' → (member ls'' → B elm) → B elm
                          end) with
                | HFirst _ ⇒ fun x _ ⇒ x
                | HNext _ _ mem' ⇒ fun _ get_mls' ⇒ get_mls' mem'
                end x (hget mls')
```

In this case the `return` clause is even more involved. Like before we pattern match on the argument used to construct the member proof. When `nil` was used to construct the member proof, then we return nothing, since the case is contradictory. In the other case we delay the recursion by requiring the recursive function to be passed in.

Then when we hit the correct element with `HFirst`, we can return it directly, and otherwise use the delayed recursion. We pass the first element of the list and use the recursive function as arguments to the last match.

This way of writing programs is very tedious, and requires some care and a lot of tricks. We will encounter more examples of dependent pattern matching later in this report .In the next section we will step back a bit, and do some more basic programs and proofs in Coq.

4.4 A first attempt at proving fusion rules

We can prove some of these properties about functions on lists directly in Coq. The proofs are done by implementing lists, map and filter in Coq and then proving the properties as theorems. We use the list definition from the standard library.

We can implement map just like one would in a functional programming language using recursion.

```
Fixpoint map (T T' : Set) (f : T → T') (ls : list T) : list T' :=
  match ls with
  | nil ⇒ nil
  | cons x ls' ⇒ cons (f x) (map f ls')
  end.
```

And now we can prove map fusion directly in Coq as a theorem. The proof is by induction over the structure of the list. The result follows directly by applying simplification and rewriting using the induction hypothesis.

Coq has support for extending the language with new notations, we can use this language feature to define a special notation for function composition.

```
Notation "f 'o' g" := (fun x ⇒ f (g x)) (at level 80, right associativity).
```

Then we can state the theorem clearly.

```
Theorem map_fusion : ∀ (T T' T'' : Set) (f : T' → T'')
                       (g : T → T') (ls : list T),
    map f (map g ls) = map (f o g) ls.
Proof. induction ls. reflexivity.
       simpl. rewrite <- IHls. reflexivity. Qed.
```

In this case the result follows directly from the induction hypothesis. This proof is actually the same as the one we did about functors in section 2.4. The case for filter fusion is only a little bit more involved. We need a boolean type for the predicates and a definition of boolean and and some basic lemmas about identities for and. They can be seen in the accompanying source code. Filter is defined as

```
Fixpoint filter (T : Set) (p : T → bool) (ls : list T) : list T :=
  match ls with
  | nil ⇒ nil
  | cons x ls' ⇒ if p x then cons x (filter p ls') else filter p ls'
  end.
```

Now we can state the fusion theorem for filter. In this case the result does not simply follow from the induction hypothesis. The base case is still pretty simple, but we have to do case analysis in the inductive case. These proofs are done by using the `destruct` tactic. Then in each case we can apply some simple fact about and, and then the result follows from the induction hypothesis.

```
Theorem filter_fusion : ∀ (T : Set) (p q : T → bool)
                          (ls : list T),
    filter p (filter q ls) = filter (fun x ⇒ and (p x) (q x)) ls.
Proof. induction ls.
  - reflexivity.
  - simpl. destruct (q t); simpl.
    * rewrite IHls. rewrite (and_true_r (p t)). reflexivity.
    * rewrite (and_false_r (p t)). rewrite IHls. reflexivity. Qed.
```

This exercise was pretty easy, but it does not really adequately encode what we are trying to do. We want to prove the properties about our own language, and not Coq. To achieve this goal we use Coq as a meta-language for encoding the constructs we want to prove properties about.

5. Encoding in Coq

In this section we will go through the Coq encoding of the language described in section 3, and all of the decisions made with respect to the representation. We will also prove some meta-theorems about the language. In particular determinism and totality of evaluation. An overview of the implementation can be found in Appendix A.

5.1 Representing bindings

A large part of formalizing programming languages is dealing with variable bindings. In our small language `let`, `map` and `filter` introduce bindings. In other programming languages, bindings are introduced with constructs like lambda abstractions and pattern matching. In this section we will go into detail about various ways of formalizing variable binding.

First order methods

One class of methods, the so-called first order methods, uses ordinary data structures like numbers and strings to encode variables.

Names

Using names for variables models the reasoning on paper directly. The variable `x` is represented as the string `x` in the meta-language. The problem with this approach is that capture avoiding substitution must be implemented as a program. In the case of two binders, where one variable shadows the other, and one wants to substitute in a term for a variable then one would need to deal with this problem explicitly.

For instance one could have the following term,

$$\texttt{let}\ x \Leftarrow y\ \texttt{in}\ (\texttt{let}\ x \Leftarrow 3\ \texttt{in}\ x + x)$$

where one would want to remove the outer `let` by substituting y for x, where one would also have to rename all instances of x in y.

Reasoning about this process mechanically quickly becomes very tedious. An incomplete formalization using names is included in the accompanying source code.

De Bruijn indices

Another technique for dealing with bindings removes the problem of shadowing by using a different method of representing bound variables [12].

This technique was first introduced by De Bruijn in 1972 for formalizing the lambda calculus. It uses natural numbers the encode the depth of the variable

relative to its binder. The previous example would be encoded as

$$\texttt{let \_} \Leftarrow \nu_2 \texttt{ in } (\texttt{let \_} \Leftarrow 3 \texttt{ in } \nu_1 + \nu_1).$$

The subscripts indicate what binder the variable is bound to. The problem here with free variables is, that the encoding is very unnatural. You have to define what it means to point into some context. This context could then be the typing context, or some evaluation context which maps variables to types or values.

Also when manipulating syntax, one has to define several operations to manipulate these indices. Some of these operations are

Lifting This operation increments indices of free variables in an expression. We lift variables whenever we do an operation under a binder.

Substitution This operation substitutes a term for a variable. This operation is going to be necessary when implementing the fusion rules.

We choose to use De Bruijn indices to encode bindings in our formalization. We will quickly go though one more popular way of representing bindings that is also used in practice.

Higher order abstract syntax (HOAS)

In the previous sections we described various first-order methods of encoding bindings. With higher-order methods one uses the binding constructs of the meta-language to encode bindings in the target language. The best known higher-order method is called higher-order abstract syntax.

This method uses a function to encode each binding. This technique is the canonical method for encoding bindings in Twelf [13], which is another framework for encoding deductive systems like programming languages. The advantage of this method is that one essentially gets substitution handed for free. A disadvantage is that it limits the way one can express bindings. If one would like to formalize dynamic scoping, HOAS would make it very difficult. Another disadvantage is, that the meta-language needs to be stronger. Standard HOAS cannot be encoded in Coq, but parametric HOAS can. Chlipala discusses this topic further in [11, chapter 17], and we will not go into further details here.

5.2 Syntax

With the choice of encoding for bindings settled, we need to encode the abstract syntax of the language. There are two major techniques of encoding syntax, intrinsic and extrinsic encoding.

Extrinsic encoding

With an extrinsic encoding, one would have separate definitions for syntax and and typing judgment. For (a subset of) our language, the syntax would be encoded in Coq in the following way

```
Inductive exp : Type :=
| evar : nat → exp
| econst : nat → exp
```

```
| enil : exp
| econs : exp → exp → exp
| emap : exp → exp → exp.
```

One would then encode the possible types for expressions in the following way

```
Inductive ty : Type :=
| TNat : ty
| TList : ty → ty.
```

And then finally one would encode the typing judgment from section 3.3 as another judgment. Then whenever one would prove some property, the proof for well-typedness would be carried around everywhere. This way of reasoning quickly becomes very tedious, and there is another technique that fixes some of these problems. We will discuss this technique in the next section. A small formalization in an untyped style can be found in the accompanying source code.

Intrinsic encoding

A method not widespread in the Coq community is using an intrinsic encoding of expressions. This encoding is also known as a Church-style encoding. The idea of this encoding is that we only assign meanings to well typed terms. An intrinsic encoding of the simply-typed lambda calculus in Coq was performed by Benton, Hur, Kennedy and McBride [14], and some of the work here will be based on that. We will also use De Bruijn-indices for variables, but De Bruijn-indices are also made a bit more complicated by the intrinsic encoding.

The main idea is to make sure that all expressions are well-typed by construction. The problem with doing this encoding is that requires more sophisticated types in the meta-language. In this case we are in luck, because the type system in Coq is very expressive, as we will see in the remainder of this project.

We start by defining a typing environment as a list of types.

```
Definition env := list ty.
```

Now the syntax is indexed by this environment. Furthermore we define the syntax as a GADT[1], such that every expression constructor has a specific type. Whenever we need to construct a variable, we need to give a proof that the variable is a member of the typing environment. Notice that whenever we extend the environment, we put the type in the beginning of the typing environment. I have left out some constructors for brevity, but they can be seen in the accompanying code.

```
Inductive exp G : ty → Type :=
| evar : ∀ t, member t G → exp G t
| econst : nat → exp G TNat
| esucc : exp G TNat → exp G TNat
| enil : ∀ t, exp G (TList t)
| econs : ∀ t, exp G t → exp G (TList t) → exp G (TList t)
| elet : ∀ t1 t2, exp G t1 → exp (t1 :: G) t2 → exp G t2
| emap : ∀ t1 t2, exp (t1 :: G) t2 → exp G (TList t1) → exp G (TList t2)
```

[1] Generalized Algebraic Data-Type

It is quite obvious that we actually have encoded the typing rules from Section 3.3 as well. Set inclusion is encoded as the `member` judgment from section 4.3.

By using the `Arguments` declaration in Coq, we can tell it to infer the typing environment and the quantified types automatically. This inference makes the encoding much nicer to work with. The declaration looks like

```
Arguments enil [G t].
```

Where `G` and `t` are now inferred automatically by Coq. Values for our language are encoded in a similar fashion.

Now as an example we can encode the program that adds one to every element of the list $[4, 5]$

$$\texttt{map}\ (\lambda x.\texttt{S}\ x)\ (\overline{4} :: \overline{5} :: \texttt{nil})$$

as

```
Example inc : exp nil (TList TNat) :=
  emap (esucc (evar HFirst)) (econs (econst 4) (econs (econst 5) enil)).
```

The `nil` in the type of `inc` denotes that the typing environment is empty. We can also ask Coq what type this expression has.

```
Coq < Check emap (esucc (evar HFirst)) (econs (econst 4) (econs (econst 5) enil)).
emap (succ (evar HFirst)) (econs (econst 4) (econs (econst 5) enil))
     : exp ?G (TList TNat)
where
?G : [ |- list ty]
```

Here `?G` is any environment. We can go even further and ask about the type of `evar (HNext HFirst)`. This term is some free variable, that points into the second element of the evaluation context.

```
Coq < Check evar (HNext HFirst).
evar (HNext HFirst)
     : exp (?x :: ?t :: ?ls) ?t
where
?t : [ |- ty]
?x : [ |- ty]
```

Now Coq infers that the type of the expression is the type of the second variable in the context.

5.3 Semantics

Now that we have defined the syntax of expressions, we need to encode the semantics as well. We have chosen to explore two different methods. One is denotational semantics, where we in this case give expressions meaning by mapping them onto regular Coq terms. The other is operational semantics, where we encode the evaluation judgment defined in section 3.4. We use a liberal interpretation of what denotational semantics means. In most programming language theory texts, denotational semantics maps terms onto domain theory, but the principle is the same. We denote expressions with a semantics from another domain.

Denotational semantics

In this section we actually write an interpreter for our language. This interpretation is what we mean when we say that we denote expressions by Coq terms.

The first thing we need is a mapping from types in our language to types in Coq. Now the dependent type system comes in handy. We define a recursive function that denotes the types. It has to be recursive, since we can have arbitrary nesting of lists.

```
Fixpoint tyDenote (t : ty) : Type :=
  match t with
  | TBool ⇒ bool
  | TNat ⇒ nat
  | TList t ⇒ list (tyDenote t)
  end.
```

This function is then used to calculate the return type for the function that interprets expressions. For brevity we have only include some of the rules, corresponding to the syntax defined in Section 5.2. Note that we return a function that expects the environment as input. Then it is easy to expand the environment as we recurse under binders. We also leverage the map function in Coq to calculate the result of a map.

```
Fixpoint expDenote G t (e : exp G t) : hlist tyDenote G → tyDenote t :=
  match e with
  | evar _ x ⇒ fun s ⇒ hget s x
  | econst n ⇒ fun _ ⇒ n
  | esucc e ⇒ fun s ⇒ S (expDenote e s)
  | enil _ ⇒ fun _ ⇒ nil
  | econs _ e1 e2 ⇒ fun s ⇒ (expDenote e1 s) :: (expDenote e2 s)
  | elet _ _ e el ⇒
    fun s ⇒ expDenote el (HCons (expDenote e s) s)
  | emap _ _ ef e ⇒
    fun s ⇒ let f := fun x ⇒ expDenote ef (HCons x s)
            in map f (expDenote e s)
  end.
```

Here we really see the value of the dependent type system. In the case for numerical constants, we can just return the natural number. In a traditional functional language, one would typically wrap the value in an abstract data type, and then one would have to unwrap it afterwards.

To show how to prove simple things using this semantic, we can try to prove that `let x ⇐ e in x` evaluates to the same value as e forall e. It turns out that this result follows directly from expanding the definition of let.

```
Lemma let_id : ∀ G t (e : exp G t) s,
    expDenote (elet e (evar HFirst)) s = expDenote e s.
Proof.
  intros. simpl. reflexivity.
Qed.
```

We are not really there yet. Just like in Section 4.4, this formalization does not adequately encode what we want. We are still just proving properties about Coq terms in some sense. Now we have just defined our own syntax for it. In the next section we will go through encoding the actual semantics in Coq.

Operational semantics

It requires a bit more care to encode the big-step semantics for our language. As a warm-up exercise we encode the semantics for append, since it does not depend on the general evaluation.

We encode evaluation judgments as inductive types. Here we define the judgment $v_1 +\!\!+ v_2 \Downarrow v$ as

```
Inductive CApp : ∀ t,
    val (TList t) → val (TList t) → val (TList t) → Prop :=
```

We specify the types here, since we also implicitly encode the typing requirements. This judgment is a relation between three lists of values, two arguments and one result. Just like in Section 3.4, we have two cases to encode.

```
| CAppNil : ∀ t (v : val (TList t)), CApp (vnil t) v v
| CAppCons : ∀ t v (v1 : val t) v2 v3,
    CApp v2 v3 v → CApp (vcons v1 v2) v3 (vcons v1 v).
```

The first case states that the result of appending any list to the empty list is that list. The second case encodes the recursive case.

To check that this encoding works, we can then prove that this append semantics is deterministic. On paper this lemma would be stated as

Lemma. *If* $v_1 +\!\!+ v_2 \Downarrow v$ *then if* $v_1 +\!\!+ v_2 \Downarrow v'$ *then* $v = v'$.

In Coq this lemma is written as

```
Lemma capp_determ : ∀ t (v1 v2 v3 v4 : val (TList t)),
    CApp v1 v2 v3 → CApp v1 v2 v4 → v3 = v4.
```

Just like on paper this lemma is proven by induction over the first derivation. The result follows directly from the induction hypothesis.

```
Proof.
  intros t v1 v2 v3 v4 H H'.
  dependent induction H; dependent destruction H'.
  - reflexivity.
  - rewrite (IHCApp v0). reflexivity. assumption.
Qed.
```

The `dependent` tactics work like the ordinary ones, but they automatically eliminate contradictory cases where the types do not match.

Things get a bit more complicated when we have to encode the evaluation judgment for expressions. We start by defining the typing context and the evaluation context

```
Definition ty_ctx := list ty.
Definition ev_ctx G := hlist val G.
```

The `hlist` is the implementation from Section 4.3. Now we can encode the evaluation judgment, $\rho \vdash e \downarrow v$, as

```
Inductive Ev : ∀ G t, ev_ctx G → exp G t → val t → Prop :=
```

Because the `map` and the `filter` judgments also depend on the evaluation judgment, we have to make this definition mutually inductive

```
with CMap : ∀ G t1 t2,
  ev_ctx G → val (TList t1) →
  exp (t1 :: G) t2 → val (TList t2) → Prop := ...
with CFilter : ∀ G t,
  ev_ctx G → val (TList t) →
  exp (t :: G) TBool → val (TList t) → Prop := ...
```

For `map` and `filter`, we also encode in the judgment that we expand the typing context with the relevant types. We will now go through the encoding of some of the non-trivial constructs.

Variables In the evaluation rule for variables, we require that the variable is a member of the evaluation context. We can use the `member` proof from the abstract syntax tree for this rule. Then we can use `hget` to retrieve the appropriate value from the context. This definition looks a lot like the rule for variables from the operational semantics.

```
| EvVar : ∀ G R t (m : member t G),
    Ev R (evar m) (hget R m)
```

We find this encoding very elegant, and it is very easy to express in Coq.

`let`-bindings Just like when we expanded the typing context when we defined the syntax, we have to expand the evaluation context. This encoding is done using `HCons` from the `hlist` implementation. We expand the context for the body by the result of evaluating the bindee. The type inference is not quite strong enough, so we write explicit types for the expressions.

```
| EvLet : ∀ (G : ty_ctx) (R : ev_ctx G) t1 t2 v1 (v2 : val t2)
           (e1 : exp G t1) (e2 : exp (t1 :: G) t2),
    Ev R e1 v1 → Ev (HCons v1 R) e2 v2 → Ev R (elet e1 e2) v2
```

`map` and `filter` Here we have to deal with the fact, that we have defined a mutually inductive semantics. The evaluation rule for `map` is pretty straight forward

```
| EvMap : ∀ (G : ty_ctx) (R : ev_ctx G) t1 t2
           (e1 : exp (t1 :: G) t2) (e2 : exp G (TList t1)) v v',
    Ev R e2 v → CMap R v e1 v' → Ev R (emap e1 e2) v'
```

This encoding means that the function parameter, `e1`, will have one bound variable, and `e2` will have some list type. We require that the list evaluates to some value. We then hand this evaluation off to the `CMap` judgment. The `map` judgment takes list values to list values. It has two cases, one for `cons` and one for `nil`

```
| CMapNil : ∀ (G : ty_ctx) (R : ev_ctx G) t1 t2
            (e : exp (t1 :: G) t2),
    CMap R (vnil t1) e (vnil t2)
| CMapCons : ∀ (G : ty_ctx) (R : ev_ctx G) t1 t2
             (e : exp (t1 :: G) t2) v1 v2 v1' v2',
    Ev (HCons v1 R) e v1' → CMap R v2 e v2' → CMap R (vcons v1 v2) e (vcons v1' v2')
```

The interesting case is the `cons` case, where we refer back to the evaluation judgment. We bind the first element of the list to the evaluation context, and we evaluate the function parameter with the variable bound. We then require that

the rest of the list is evaluated with `CMap` as well. This requirement corresponds to a recursive call in a functional programming language.

`filter` is implemented in a similar way. When implementing denotational semantics, we proved a property about wrapping an expression in a let, and we can prove the same property using operational semantics.

```
Lemma let_id_sound2 : ∀ t G (R : ev_ctx G) (e : exp G t) (v1 v2 : val t),
    Ev R e v1 → Ev R (elet e (evar HFirst)) v2 → v1 = v2.
Proof. intros.
       apply EvLet with (e2 := (evar HFirst)) (v2 := v1) in H.
       apply (ev_determ H) in H0.
       exact H0. constructor. Qed.
```

By using an intrinsic encoding, we actually get some proofs for free. For instance we know that evaluation is type preserving, this type-reserving encoding means that we know that the following lemma holds.

Lemma. If $\Gamma \vdash e : \tau$, $\rho \vdash e \downarrow v$ and Γ and ρ agree then $v : \tau$. Γ and ρ agree if for every $x : \tau \in \Gamma$ then there is a $x \downarrow v \in \rho$ such that $v : \tau$.

We do not have to deal with this agreement of typing and evaluation contexts, because the evaluation context is indexed by the typing context, so we have this agreement by construction.

We are almost ready to do some proofs about our language. We are still missing one small detail. It turns out that the induction principle that Coq generates for this inductive type is not strong enough to prove interesting properties. To solve this problem we need to write our own induction principle.

5.4 Induction principles

To understand how induction principles work in Coq, we step back a bit, and try to understand a simple induction proof in depth. Say, we wanted to prove that $\forall n \in \mathbb{N}.n = n + 0$. This is easily done in Coq by induction on n.

```
Theorem plus_n_0 : ∀ n : nat, n = n + 0.
Proof. induction n. reflexivity.
       simpl. rewrite <- IHn. reflexivity. Qed.
```

Under the hood, Coq uses an induction principle for natural numbers to do this induction. We can inspect this induction principle by printing `nat_ind`.

```
nat_ind : ∀ P : nat → Prop,
  P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

This is actually a function, that when given a predicate from natural numbers to propositions, a case for zero, and an inductive case gives us a theorem in return. We can apply this principle as a regular function.

```
Theorem plus_n_0' : ∀ n : nat, n = n + 0.
Proof. apply (nat_ind (fun n : nat ⇒ n = n + 0)).
  - reflexivity.
  - intros. simpl. rewrite <- H. reflexivity. Qed.
```

We could also have given proofs for the base case and the inductive cases directly as arguments to `nat_ind`, but we just chose to use tactics instead to generate the proof terms. There is also a shortcut for using other induction principles. Then one writes

```
induction n using nat_ind with (P := fun n ⇒ n = n + 0)
```

instead. This short-hand gives some cleaner names for the induction hypothesis. Now with further understanding of induction in Coq, we can do some proofs about our language.

A very simple property that we might want to prove about our language is that evaluation is deterministic. We have already proven that the append function is deterministic, and we will prove determinism for evaluation of expressions in the same way.

```
Theorem ev_determ : ∀ G t (R : ev_ctx G) (e: exp G t) v1 v2,
    Ev R e v1 → Ev R e v2 → v1 = v2.
```

If we proceed by induction on the first evaluation derivation, we hit the case for `map`. We can try to write a lemma for `map` in the same way as we did for append, but we will fail, since determinism of `map` depends on determinism of general evaluation. The generated induction principle does not capture this mutual relationship.

Coq luckily has support for custom induction principles, and we can write one using the `Scheme` command.

```
Scheme Ev_mut := Induction for Ev Sort Prop
    with CMap_mut := Induction for CMap Sort Prop
    with CFilter_mut := Induction for CFilter Sort Prop.
```

This definition states that we want Coq to generate a mutual induction principle for all three judgments. We can inspect the induction principle to see what Coq generates.

```
Ev_mut : ∀
  (P : ∀ (G : list ty) (t : ty) (e : ev_ctx G) (e0 : exp G t) (v : val t),
       Ev e e0 v → Prop)
  (P0 : ∀ (G : list ty) (t1 t2 : ty) (e : ev_ctx G)
          (v : val (TList t1)) (e0 : exp (t1 :: G) t2) (v0 : val (TList t2)),
        CMap e v e0 v0 → Prop)
  (P1 : ∀ (G : list ty) (t : ty) (e : ev_ctx G) (v : val (TList t))
          (e0 : exp (t :: G) TBool) (v0 : val (TList t)),
        CFilter e v e0 v0 → Prop), (* All cases *) P G t e e0 v e1.
```

This principle means that we have to specify three induction hypothesis, one for each judgment. Then Coq can give us a theorem P that holds for all expressions. By further inspecting the induction principle we can see that in the case for `map` we get `P0` as a premise, which is exactly what we want.

We also have to prove that `P0` holds for `CMap` and that `P1` holds for `CFilter`. We can now use this induction principle in our proof for determinism.

```
Proof.
  intros.
  generalize dependent v2.
  induction H using Ev_mut.
```

Coq complains that it cannot find instances for `P`, `P0` and `P1`. We need to supply them manually. We just has to check the type signature for the induction hypothesis and fill in the goal for `Prop`.

```
induction H using Ev_mut with
      (P := fun (G : list ty) (t : ty) (R : ev_ctx G)
                (e : exp G t) (v : val t)
                (ev : Ev R e v) ⇒ ∀ v2, Ev R e v2 → v = v2)
      (P0 := fun (G : list ty) (t1 t2 : ty) (R : ev_ctx G)
                 (v : val (TList t1)) (e : exp (t1 :: G) t2) (v0 : val (TList t2))
                 (ev : CMap R v e v0) ⇒ ∀ v2, CMap R v e v2 → v0 = v2)
      (P1 := fun (G : list ty) (t : ty) (R : ev_ctx G)
                 (v : val (TList t)) (e : exp (t :: G) TBool) (v0 : val (TList t))
                 (ev : CFilter R v e v0) ⇒ ∀ v2, CFilter R v e v2 → v0 = v2).
```

Notice that in `P0` and `P1` we write a possible induction hypothesis for determinism of `CMap` and `CFilter` respectively. Then we get determinism for `CMap` and `CFilter` as an hypothesis when doing the proof for `Ev`. Now the proof can be completed. This proof can be seen in the accompanying code.

We also prove totality of expression evaluation. Totality means that every expression evaluates to a value. This is not as easy as determinism, and the proof can be seen in the accompanying source code. The mutually inductive semantics makes things a bit complicated again. These proofs are done on the syntax of e. We define two helper lemmas

```
Lemma cmap_total : ∀ G t t' (R : ev_ctx G) (e : exp (t :: G) t') (vs : val (TList t)),
      (∀ v, ∃ v', Ev (HCons v R) e v') → ∃ vl, CMap R vs e vl.

Lemma cfilter_total : ∀ G t (R : ev_ctx G) (e : exp (t :: G) TBool) (vs : val (TList t)),
    (∀ v, ∃ vb, Ev (HCons v R) e vb) → ∃ vl, CFilter R vs e vl.
```

In essence they assume totality for expressions, and use that to prove totality of the map and filter judgments. Now totality for expressions follows.

```
Theorem ev_total : ∀ G t (R : ev_ctx G) (e : exp G t),
    ∃ v, Ev R e v.
```

This theorem turns out to be useful when dealing with the short circuiting and operator. It will probably not be the case that all expressions evaluate to a value if we add function calls and recursion to the language.

We can now proceed to actually implementing and proving properties about the fusion rules.

6. Correctness of fusion rules

We have defined our language and its semantics. It is now time to prove the fusion rules correct. We first have to define the rules in Coq.

6.1 Defining the rules in Coq

Since we have decided to work with intrinsically typed syntax, we have to work a bit harder to define the rules. With the untyped syntax, one could just rewrite the abstract syntax tree, but with intrinsic types, implementation of the rules and proofs of type-preservation must be given at the same time.

Substitution

As described in section 2, we need to define a notion of substitution for our language to be able to formalize map fusion. In contrast to formalizations of the lambda calculus, we do not use substitution in the evaluation rules. The way we defined substitution is heavily influenced by [14].

A substitution is defined as a function that given a proof of membership in a typing context `G` gives us an expression typed in a context `G'`.

```
Definition Sub G G' := ∀ t, member t G → exp G' t.
```

The idea is that this substitution is a list of substitutions mapping variables at each index to some expression. We define the identity substitution as the substitution that just maps variables to themselves.

```
Definition idSub {G} : Sub G G := @evar G.
```

Now `consSub` extends a substitution by mapping the next variable to an expression. We use the `Program` tactic to write this function, because it simplifies pattern matching on GADTs.

```
Program Definition consSub {G G' t} (e:exp G' t) (s:Sub G G') : Sub (t::G) G' :=
  fun t' (v: member t' (t::G)) ⇒
    match v with
    | HFirst _ ⇒ e
    | HNext _ _ v' ⇒ s _ v'
    end.
```

We define a shorthand for defining a substitution.

```
Notation "{| e ; .. ; f |}" := (consSub e .. (consSub f idSub) ..).
```

In section 5.1 we stated that lifting is one of the operations that one would like to define when using De Bruijn indices. Instead of lifting, we define a more general notion of renaming instead.

```
Definition Ren G G' := ∀ t, @member ty t G → @member ty t G'.
```

Which means that a typing context can be renamed if all variables are members of both contexts. With this definition we can define lifting of renaming, renaming of expressions, shifting of expressions, shifting of substitutions, and finally substitutions in expressions. All of the helper functions can be found in the accompanying source code. Now we can define substitutions in expressions by recursively applying the substitution on all variables. We have not included all cases in the report.

```
Fixpoint subExp G G' t (s : Sub G G') (e : exp G t) :=
  match e with
  | evar _ x ⇒ s _ x
  | econst n ⇒ econst n
  | esucc e ⇒ esucc (subExp s e)
  | enil _ ⇒ enil
  | econs _ e1 e2 ⇒ econs (subExp s e1) (subExp s e2)
  | elet _ _ e1 e2 ⇒ elet (subExp s e1) (subExp (subShift s) e2)
  | emap _ _ e1 e2 ⇒ emap (subExp (subShift s) e1) (subExp s e2)
  end.
```

Notice that we shift the substitution whenever we have bindings.

Function composition with substitution

Since we do not have function objects, we have a different notion of function composition. A function is an expression with its free variable at the first index. A function $f : t_1 \longrightarrow t_2$ is encoded as an expression `f : exp (t1 :: G) t2` for some `G`. For some other function `g : exp (t2 :: G) t3`, we have that the composition is a new expression `compose f g : exp (t1 :: G) t3`. We try to write this function in Coq.

```
Definition compose t1 t2 t3 G
    (f : exp (t1 :: G) t2)
    (g : exp (t2 :: G) t3) : exp (t1 :: G) t3 :=
  subExp {| f |} g.
```

Coq complains, because it expects that `g` has type `exp (t2 :: t1 :: G) t3`. We need to weaken this expression to be typed in a larger typing context. We can easily implement this function as well.

```
Definition shift2Exp G t1 t2 t3 : exp (t2 :: G) t3 → exp (t2 :: t1 :: G) t3.
```

Now we can define composition

```
Definition compose t1 t2 t3 G
    (f : exp (t1 :: G) t2)
    (g : exp (t2 :: G) t3) : exp (t1 :: G) t3 :=
  subExp {| f |} (shift2Exp _ g).
```

We are now ready to define the fusion transformations.

Program transformations

The first fusion rule that we try to define is filter fusion. This rule is easier than map fusion because we do not need substitution to implement to implement this rule. We try to define it naively just as you would in a language like ML.

```
Definition filter_fusion' t G (e : exp G t) : exp G t :=
  match e with
  | efilter t' p el ⇒ match el with
                      | efilter t' q body ⇒ efilter (eand p q) body
                      | _ ⇒ e end
  | _ ⇒ e end.
```

Coq rejects this definition, because it cannot infer that `e` has the same type as the second match expression. we need to explicitly convince Coq of this equality by giving it a proof for the fact. This proof requires a few tricks. One way of solving this problem is by constructing the program interactively.

The `refine` tactic makes it possible to give a term with holes to be filled later. We start by giving the type

```
Definition filter_fusion t G (e : exp G t) : exp G t.
```

Then we give a term that looks like the previous one, but with a few changes. We use a trick where we defer argument binding in a pattern match until later. We defer the argument so that it can be referenced in the return clause of the dependent pattern match.

```
refine
  (match e in exp _ t' return exp G t' → exp G t' with
   | efilter te p em ⇒ (fun lt (em' : exp G lt) ⇒
                         match em' in exp _ lt return (TList te = lt → _) with
                         | efilter t' q eb ⇒ fun Heq _ ⇒ efilter (eand _ q) eb
                         | _ ⇒ fun _ e ⇒ e
                         end) (TList te) em _
   | _ ⇒ fun e ⇒ e
   end e).
```

We annotate the match expression with a different return type. The interesting case is the inner match. We know intuitively that the type `te` must be the same as `t'` in the inner match. We solve this problem by requiring a proof of their equality to be passed in. Furthermore `p` does not have the correct type, therefore we leave it as a hole.

Now Coq presents us with 2 goals to prove. We need to give a term of type `exp (t' :: G) TBool` but `p` has type `exp (te :: G) TBool`. As a premise we have that `TList te = TList t'`. Then it follows by injectivity that `p` has the correct type.

```
injection Heq. intros. rewrite <- H. exact p.
```

Now Coq presents us with the last goal. We need to show that `TList te = TList te`. This proof is trivial. Coq would have been equally satisfied with `q` instead of `p`. It would have the correct type, but the resulting program would not have the expected behavior. If we go back to thinking about propositions as types, then the type of `filter_fusion` is the proposition, and the implementation is the proof of the proposition. If we imagine a proof with `p` and a proof with `q`, then they both would have been valid proofs, but if we were to run them as programs (and we are) then, only one of them would have the expected behavior.

This kind of programming really blurs the distinction between programs and proofs. We can inspect the resulting program by using `Print`, but the term is way too large to fit in the report.

We can define map fusion using the `compose` function that we defined in the previous section. It looks similar to `filter_fusion`, and the implementation can be seen in the accompanying source code.

6.2 Proofs

Because we have defined the transformations on intrinsically typed syntax, we get type preservation for free. One could state this on paper as

Lemma If $\Gamma \vdash e : \tau$ then $\Gamma \vdash \texttt{map\_fusion}(e) : \tau$.

In this section we attempt to prove filter fusion and map fusion correct with respect to both semantics. We start with the denotational semantics.

Denotational semantics

The easiest proof is the one about filter fusion. We need to prove that for all expressions, if we apply `filter_fusion` to it, then we get the same result.

```
Theorem filter_fusion_sound t (e : exp nil t) s:
    expDenote e s = expDenote (filter_fusion e) s.
```

We prove this theorem by case analysis, and in essence the proof looks a lot like the one we did in section 4.4.

The main lemma that the correctness of map fusion relies on is that our substitution is correct. We state this lemma by proving that the result of evaluating the composition is the same as putting the result of evaluating one function into the environment and then evaluating the other function.

```
Lemma compose_sound : ∀ t1 t2 t3 G R (f: exp (t1 :: G) t2) (g: exp (t2 :: G) t3) v1,
    expDenote g (HCons (expDenote f (HCons v1 R)) R) =
    expDenote (compose f g) (HCons v1 R).
```

This proof has some issues. The way we defined composition is very complex, and we have a lot of definitions on top of each other. We did not manage to prove that this composition is sound.

If we assume the composition is sound, then it follows directly that map fusion is sound.

```
Theorem map_fusion_sound t (e : exp nil t) s :
    expDenote e s = expDenote (map_fusion e) s.
```

The proof strategy is the same as for filter fusion, and the full proofs can be found in the accompanying source code.

Operational semantics

When proving properties about the operational semantics, we have to deal with the fact that it is defined as a relation, and not a total evaluation function. The proofs are going to have a different structure, where we assume that an expression evaluates to a value, and then we need to prove that when the transformation is applied to the expression, it evaluates to the same thing. The theorem for filter fusion is defined as follows

```
Theorem filter_fusion_sound2 : ∀ t G (R : ev_ctx G) (e : exp G t) (v : val t),
        Ev R e v → Ev R (filter_fusion e) v.
```

The theorem for filter fusion relies on two lemmas. One states that if the right conjunct evaluates to false, then the entire and expression evaluates to false.

```
Lemma and_r_false : ∀ G (R : ev_ctx G) (e1 e2 : exp G TBool),
    Ev R e2 vfalse → Ev R (eand e1 e2) vfalse.
```

And another lemma states that `CFilter` can be fused.

```
Lemma cfilter_fusion : ∀ t G (R : ev_ctx G) (e1 e2 : exp (t :: G) TBool)
                         (v1 v2 v3 : val (TList t)),
  CFilter R v1 e1 v2 → CFilter R v2 e2 v3 → CFilter R v1 (eand e2 e1) v3.
```

Notice how cleanly the lemmas and theorems are stated. The theorem for map fusion is stated as follows.

```
Theorem map_fusion_sound2 : ∀ t G (R : ev_ctx G) (e : exp G t) (v : val t),
        Ev R e v → Ev R (map_fusion e) v.
```

Just like before we delegate most of the work to two lemmas. One states that composition is sound

```
Lemma compose_sound2 : ∀ t1 t2 t3 G (R : ev_ctx G)
                         (e1 : exp (t1 :: G) t2) (e2 : exp (t2 :: G) t3)
                         (v1 : val t1) (v2 : val t2) (v3 : val t3),
    Ev (HCons v1 R) e1 v2 → Ev (HCons v2 R) e2 v3 →
    Ev (HCons v1 R) (compose e1 e2) v3.
```

And another lemma to state that `CMap` can be fused

```
Lemma cmap_fusion : ∀ t1 t2 t3 G (R : ev_ctx G)
                      (e1 : exp (t1 :: G) t2) (e2 : exp (t2 :: G) t3)
                      (v1 : val (TList t1)) (v2 : val (TList t2)) (v3 : val (TList t3)),
    CMap R v1 e1 v2 → CMap R v2 e2 v3 → CMap R v1 (compose e1 e2) v3.
```

Just like in the proofs using denotational semantics, we did not manage to proof that composition is sound. So if we assume that our notion of composition is sound, then we know that both map and filter fusion are sound with respect to both semantics. The full proofs can be found in the accompanying source code.

6.3 Code extraction

The final thing we want to explore is the possibility of using Coq's code extraction capabilities, and the possibility of using the extracted code in a realistic compiler. In [15] Swiertstra describes some of the challenges he faced when extracting code from Coq and tried to incorporate it into an existing Haskell code base.

The extraction process generates code for all functions and data types. The problem arises when we need to interface with non-Coq code. Coq gives us the possibility to replace certain definitions with user-defined ones. For instance we can replace a generated boolean type with the equivalent Haskell type.

```
Extract Inductive bool ⇒
    "Bool" ["True" "False"].
```

But we could just as well have written

```
Extract Inductive bool ⇒
    "Bool" ["False" "True"].
```

The code would still type check, but the behaviour would be radically different.

In our case we can have a lot of *impedance-matching* problems. Our language encoding is radically different from one that you would use in a realistic compiler. We would need to write intermediate code to convert between different encodings. If the compiler that the code would be embedded in does not use De Bruijn indices, then one would need to convert named variables to De Bruijn indices.

We try to generate Haskell code for the map fusion rule by writing

```
Extraction Language Haskell.
Extraction map_fusion.
```

Then Coq outputs the following Haskell code

```
map_fusion :: Ty -> (List Ty) -> Exp -> Exp
map_fusion _ g e =
  case e of {
   Emap t2 t3 g0 em ->
    case em of {
     Emap t4 t2' f eb -> Emap t4 t3 (compose t4 t2' t3 g f
                                     (eq_rec_r t2' (\g1 _ _ -> g1) t2 g0 em __)) eb;
     _ -> e};
   _ -> e}
```

We see that Coq did not manage to remove all equality proofs. The difficulty of defining transformations on intrinsically typed syntax also gave problems for code extraction. If we inspect the generated data types, we also see that the extractor did not generate a GADT in Haskell even though it is supported. Most projects that use extraction use OCaml and not Haskell. We can see if the OCaml extractor generates simpler code.

```
let map_fusion _ g e = match e with
| Emap (_, t3, g0, em) ->
  (match em with
   | Emap (t4, t2', f, eb) -> Emap (t4, t3, (compose t4 t2' t3 g f g0), eb)
   | _ -> e)
| _ -> e
```

This extracted code looks simpler, but the difficulty is still to make sure that all expressions are well-typed when fed into the extracted code.

These examples illustrate some of the problems with a more unnatural encoding. Some of the proofs are tightly coupled with definitions and hard to extract.

7. Related work

In this section we will explain how fusion is performed in other compilers, and how some of the concepts used in our work can be used in other ways with array programming.

7.1 Other fusion rules in Futhark

In Futhark [6] some other SOACs and fusion rules are also used. Futhark is a data-parallel programming language designed for running on GPUs. I addition to the ordinary SOACs like map, reduce and filter. Futhark also has support for streaming SOACs. The rationale for streaming SOACs is that GPUs have a large, but fixed amount of parallelism that can be exploited. Whenever that amount of parallelism is exceeded, it can be beneficial to sequentialize parts of the computation.

The `stream_map` SOAC recieves a number of input arrays. It then produces an arbitrary amount of output arrays with the mapped function applied on each output array. Futhark then also defines a rule for introducing this SOAC.

$$\texttt{map}\, f \equiv \texttt{stream\_map}\,(\texttt{map}\, f) \tag{7.1}$$

and some fusion rules, for instance

$$\texttt{stream\_map}\, f \circ \texttt{map}\, g \equiv \texttt{stream\_map}\,(f \circ \texttt{map}\, g). \tag{7.2}$$

Futharks `stream_red` generalises `stream_map` by allowing each chucked array to produce an additional result that can be passes from chunk to chunk. The article explains how this construct is very useful when implementing k-means in a data-parallel way. This SOAC also has some fusion rules that can allow for certain optimizations.

`partial_map`

In our own work we dealt with filter fusion. There is also a rule to push a filter through a map

$$\texttt{filter}\, p \circ \texttt{map}\, f \equiv \texttt{map}\, f \circ \texttt{filter}\,(p \circ f) \tag{7.3}$$

This can be useful when fusing larger programs. The problem with this is that we repeat the computation of f. We can try to solve this by introducing a new SOAC,

$$\texttt{partial\_map} : \forall \alpha\beta, (\alpha \to \beta\ \texttt{option}) \to [\alpha] \to [\beta]. \tag{7.4}$$

This combinator has some really nice properties, for instance we can fuse it with a map

$$\texttt{partial\_map}\, f \circ \texttt{map}\, g \equiv \texttt{partial\_map}\,(f \circ g) \tag{7.5}$$

eliminating the repeated computation that we had before. We can introduce `partial_map` from a filter with the following rule

$$\texttt{filter}\ p \equiv \texttt{partial\_map}\ (\lambda x.\texttt{if}\ p\ x\ \texttt{then}\ \texttt{SOME}\ x\ \texttt{else}\ \texttt{NONE}). \tag{7.6}$$

This construct is considered for implementation in Futhark. And can possibly make some programs with filter more efficient.

7.2 Deforestation in GHC

In functional programming languages, programs are often composed of many small functions that use lists as intermediate data structures. Programming languages like Haskell and SML also include large libraries of list-manipulating functions that are easily combined to form more complicated programs.

One problem with this approach is that one has to allocate all these intermediate data structures, and these allocations may cause inefficiencies. This problem is solved by a process called deforestation. Gill, Launchbury and Jones [16] describe a simple method for deforestation that was implemented in the Glascow Haskell compiler in 1993.

They standardize the way lists are produced, and the way lists are consumed in a program.

Consuming lists The observation is, that many programs that consume lists can be written using the higher order function `foldr`.

$$\texttt{foldr}\ (\oplus)\ \texttt{id}\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\cdots (x_n \oplus \texttt{id}))) \tag{7.7}$$

Producing lists They also observe that building a list can be abstracted using a `build` function

$$\texttt{build}\ g = g\ \texttt{cons}\ \texttt{nil} \tag{7.8}$$

Now the `foldr/build` rule states that

$$\texttt{foldr}\ k\ z\ (\texttt{build}\ g) = g\ k\ z \tag{7.9}$$

From the article, a formula for `map` is given using both `build` and `foldr`. This time we will write it in Haskell syntax.

```
map f xs = build (\c n -> foldr (\a b -> c (f a) b) n xs)
```

Using `foldr/build` rule, and simple beta reduction we can perform map fusion.

```
map f (map g xs)
-- Unfolding map
= build (\c n -> foldr (\a b -> c (f a) b) n (map g xs)) =
-- Unfolding map
= build (\c n -> foldr (\a b -> c (f a) b) n
    (build (\c n -> foldr (\a b -> c (g a) b) n xs)))
-- Applying foldr/build rule
= build (\c n -> (\c n ->
    foldr (\a b -> c (g a) b) n xs) (\a b -> c (f a) b) n)
-- Beta reduction
= build (\c n -> foldr (\a b -> (\a b -> c (f a) b) (g a) b) n xs)
-- Beta reduction
= build (\c n -> foldr (\a b -> c (f (g a)) b) n xs)
-- Refold map
= map (\a -> f (g a)) xs = map (f . g) xs
```

This method works well for some programs, but not all programs on lists can be easily written using `foldr/build`. The article mentions that especially programs involving `zip` are a lot harder.

Correctness of the rule is justified in the article by a concept called free theorems. We will not go into further detail about free theorems here.

7.3 Dependently typed arrays

The canonical example of dependently typed programming is length indexed lists. In Coq this looks like

```
Inductive vect A : nat → Type :=
| Nil : vect A 0
| Cons : ∀ n, A → vect A n → vect A (S n).
```

If we let Coq infer `A` and `n` then we can create a list by writing

```
Coq < Check (Cons 1 (Cons 2 (Cons 3 Nil))).
Cons 1 (Cons 2 (Cons 3 Nil))
     : vect nat 3
```

Coq now tells us that this lists has length 3. Now one can use this inductive type to for instance write a zip function that expects two lists of the same length, and then have this property statically checked.

This concept is taken even further in [17] where Trojahner and Grelck propose a type system for static verification of array programs. Their type system make it possible to define for instance shape-generic array addition by extending scalar addition. Their type system is a weaker form of dependent types that is used to express constraints between array ranks, shapes, and values.

7.4 Vellvm

In the introduction we mentioned CompCert as a large development in the verified compiler community. Another large development is the Vellvm project [18]. It is a Coq formalization of the semantics of a subset of the LLVM intermediate language. It is intended for verification of LLVM software.

When writing a LLVM-based compiler, one writes a translator from the high level language to the LLVM intermediate language. Then the LLVM tools provide an array of transformations including optimizations, program transformations and static analyses. Then the resulting LLVM intermediate code can be translated to the target architecture. The goal of Vellvm is then to formalize this intermediate language to be able to verify program transformations with respect to both static semantics and operational semantics. Vellvm provides several operational semantics, where the most general is a small-step, non-deterministic evaluation relation.

$$\mathit{config} \vdash S \twoheadrightarrow S' \tag{7.10}$$

Where *config* is a collection of function tables, globals and modules. S and S' are machine states which consist of memory M and stack frames $\overline{\Sigma}$.

They have also used the code extraction capabilities of Coq to extract an interpreter for LLVM code that can then be used for running the LLVM test suite. Because a nondeterministic relation cannot be executed directly, they have defined another deterministic semantics which they prove similar to the small-step semantics and then extract.

With the Vellvm framework they formalize SoftBound, which is a hardening transformation which protects C programs from memory safety violations. They then use Vellvm to verify the correctness of SoftBound with respect to the operational semantics of LLVM. This is very close to what we did with our fusion rules. We checked that they preserved the semantics of all source programs.

The Vellvm project is closer to what we want to do than CompCert, but Vellvm is a very large project which is tens of thousands of lines of Coq code, and on top of that it uses code from CompCert as well.

8. Conclusion

In this work we ended up formalizing a programming language with second-order array combinators. We formalized it in Coq using an intrinsically typed syntax and De Bruijn indices for binders. We formalized both a denotational semantics and a more traditional big-step operational semantics. Both of these use an evaluation context to simulate the use of a variable table. We also formalized some very basic fusion rules on this syntax. We started work on proving that these fusion rules are correct, but we did not complete it.

We are not really happy with the way that composition is defined. If one expands all the relevant definitions, the resulting code is very large. This makes it difficult to figure out exactly what lemmas are needed.

Formalizing programming language theory is hard work. It takes a large amount of trial and error to separate good ideas from bad ideas. The literature on the subject is hard to understand, and they often formalize the simply-typed lambda calculus, and a large part of this work is abstracting away the parts that we needed for this formalization. We evaluate some of the choices made in this project.

8.1 Using Coq

We think that Coq has been a good choice for this project. It is very mature, and there are many resources available for learning idiomatic Coq patterns. There has been many larger developments using Coq, and research on new techniques for working with Coq is still underway.

One problem is that Coq has a pretty steep learning curve, and working with dependent types is especially difficult. Maybe this will become better in the future, but at the moment we find it suboptimial. Some of these problems can be solved using more advanced tactics, but we still had major problems.

8.2 Choice of encoding

One major advantage of the chosen encoding is how cleanly the theorems are stated. Since type-preservation is implicit, the only thing that is stated is the semantic properties that we want to prove.

The problem in a lot of the proofs is that the definitions are very large. When dealing with both intrinsically typed syntax and De Bruijn indices, you juggle around with a lot of equality proofs. Some of these problems may come from lack of experience and non-idiomatic Coq usage.

It is also very tedious to do inductive proofs with mutually inductive definitions. We had a problem when doing proofs over the syntax about some properties of evaluation relation.

8.3 Future work

The first task would be to finish the two admitted proofs about `compose`. This would involve a lot of proofs about the way that substitution is defined. It could also be interesting to look into proof automation, because it would be really handy to just define a fusion rule, and then the proofs could follow almost automatically. Adam Chlipala talks a lot about proof automation is his book [11].

It would obviously make a lot of sense to add more language features to make it possible to reason about more complicated constructs like pairs and multidimensional arrays.

It could also be beneficial to investigate whether other encodings are easier to work with. In this work we only really focused on one type of encoding, but a lot of different encodings are in use on various projects.

Another direction could be to try to investigate whether other proof assistants/theorem provers would make things simpler. Working with dependent types is a lot easier in Agda because it relies on Axiom K, which makes assumption about identity proofs and simplify dependent pattern matching significantly.

Bibliography

[1] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[2] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ml. In *ACM SIGPLAN Notices*, volume 49, pages 179–191. ACM, 2014.

[3] Kenneth E Iverson. A programming language. 1962.

[4] Timothy A. Budd. An apl compiler for a vector processor. *ACM Trans. Program. Lang. Syst.*, 6(3):297–313, July 1984.

[5] Martin Elsman and Martin Dybdal. Compiling a subset of apl into a typed intermediate language. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, page 101. ACM, 2014.

[6] 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. 2017.

[7] Richard S Bird. An introduction to the theory of lists. In *Logic of programming and calculi of discrete design*, pages 5–42. Springer, 1987.

[8] Richard Bird and Oege De Moor. The algebra of programming. In *NATO ASI DPD*, pages 167–203, 1996.

[9] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.

[10] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

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

[12] Stefan Berghofer and Christian Urban. A head-to-head comparison of de bruijn indices and names. *Electronic Notes in Theoretical Computer Science*, 174(5):53–67, 2007.

[13] Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *International Conference on Automated Deduction*, pages 202–206. Springer, 1999.

[14] Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. Strongly typed term representations in coq. *Journal of automated reasoning*, 49(2):141–159, 2012.

[15] Wouter Swierstra. Xmonad in coq (experience report): Programming a window manager in a proof assistant. *SIGPLAN Not.*, 47(12):131–136, September 2012.

[16] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[17] Kai Trojahner and Clemens Grelck. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming*, 78(7):643–664, 2009.

[18] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *ACM SIGPLAN Notices*, volume 47, pages 427–440. ACM, 2012.

A. Source code overview

In this appendix we will give an overview of the accompanying source code and a guide to running it. The newest version of the code can be found on `https://github.com/christiankjaer/soac-coq`.

A.1 Included files

`hlist.v` This file contains the heterogeneous list implementation described in section 4.3.

`ling_syntax.v` This file contains the complete specification of the syntax for both expressions and values using intrinsic types and De Bruijn indices.

`ling_semantics.v` This file contains the both the denotational semantics and the big-step operational semantics for the language as well as the induction principle.

`ling.v` This file contains all the program transformations, custom tactics, and all the proofs.

`simple/proofs.v` This file contains the proofs mentioned in section 4.4 and definitions of `foldr` and `build` from section 7.2.

`untyped/ling.v` This file contains the beginning of a formalization using names and untyped syntax.

`untyped/debruijn.v` This file contains the beginning of a formalization using De Bruijn indices and untyped syntax. It uses the Autosubst library[1]

A.2 Running the code

The code is tested with Coq 8.6[2]. The simplest way to run the code is to run

```
$ coqtop -l ling.v
# Lots of output
```

One can step through the proofs by using either Proof General[3] or CoqIDE (included in the Coq distribution). Using one of these tools is highly recommended, because using the Coq system is a highly interactive experience.

[1] `https://www.ps.uni-saarland.de/autosubst/`
[2] `https://coq.inria.fr/download`
[3] `https://proofgeneral.github.io/`