CS 4860 Final Project:

Correct Abstract Type Implementations by Commutativity

Ethan Koenig (etk39)

December 6, 2016

Abstract

This report describes and formalizes a paradigm for proving implementations of an abstract type correct. This paradigm, known as the commutativity paradigm, is then instantiated using the Coq proof assistant.

1 Introduction and Background

Abstract types play a foundational role in computer science and software development. Effective use of abstract types makes code more modular, reusable, adaptable, and comprehensible.

In many programming languages, an abstract type's interface is expressed as a collection of values that every implementation of the abstract type must provide; we will refer to these values as the abstract type's "provided values". As a running example, consider the following abstract type for last-in-first-out stacks (in OCaml-like pseudo-code):

```
abstract interface LifoStack<T> {
    empty : LifoStack<T>
    push : LifoStack<T> -> T -> LifoStack<T>
    pop : LifoStack<T> -> option (T * LifoStack<T>)
    size : LifoStack<T> -> int
}
```

Figure 1: An abstract type for last-in-first-out stacks.

However, there is more to an abstract type than its provided values. Users of an abstract type expect not only a collection of correctly-typed values, but also for these values to adhere to some established semantics. These semantics can be represented as a collection of "correctness properties" that an implementation's provided values should satisfy. An abstract type's semantics are rarely formalized as such, but the semantics which correctness properties represent are nonetheless an integral part of the abstract type.

For instance, an acceptable implementation of `LifoStack<T>` must not only have well-typed `empty`, `push`, `pop` and `size` values, but these values must exhibit last-in-first-out behavior. Intuitively, `pop l` should return the last element `push`ed onto `l`, or **`None`** if `l` is empty. An implementation where `pop` always returns **`None`** is not a valid implementation of `LifoStack<T>`. Instead, a valid implementation of `LifoStack<T>` should satisfy the following correctness properties:

```
pop empty = None
pop (push l x) = Some (x, l)  (* forall l, x *)
size empty = 0
size (push l x) = 1 + size l  (* forall l, x *)
```

Figure 2: Correctness properties for `LifoStack<T>`

We will refer to a collection of correctness properties as sound if (and only if) the collection guarantees correct behavior. For instance, any implementation of `LifoStack<T>` satisfying the four above properties will exhibit last-in-first-out behavior, so the collection of properties is sound. Removing any one of these properties causes the collection to become unsound.

Unfortunately, it can be difficult to devise a sound set of correctness properties for an abstract type. For example, it takes a bit of reasoning to convince oneself that the four above correctness properties for `LifoStack<T>` are sound, and `LifoStack<T>` is a very simple abstract type. Ensuring that a set of correctness properties is sound becomes even more difficult as the number provided values grows.

In contrast, it is often very straightforward to convince oneself that a simple, canonical implementation of an interface is correct, even if formally expressing correctness is difficult! Furthermore, it is often very straightforward to prove properties about simple implementations. For instance, it is rather clear that the following implementation of `LifoStack<T>` is correct, and all the above correctness properties hold trivially.

```
implementation NilConsLifoStack<T> : LifoStack<T> {
    empty = []
    push stack elem = elem::stack
    pop stack = match stack with
                | [] -> None
                | head::tail -> Some (head, tail)
    size stack = match stack with
                 | [] -> 0
                 | _::tail -> 1 + size tail
}
```

Figure 3: A “canonical” implementation of `LifoStack<T>` using nil-cons lists.

Unfortunately, the same cannot be said of more complicated implementations. For an arbitrary implementation of an abstract type, the only way to be sure that the implementation is correct is to know that it satisfies a sound collection of correctness properties. However, proving properties of arbitrary implementations can be difficult, and we may not even be sure that the collection of correctness properties we prove about the arbitrary implementation is sound.

At the same time, these non-canonical implementations are valuable, since they may have advantages over the canonical implementation, such as better performance, more security, or higher parallelizability. We find ourselves wanting both the ease of proving correctness that the canonical implementation affords, and the advantages that more complicated implementations offer, but only being able to have one.

2 Commutativity

The commutativity paradigm is a way to keep all of the advantages that arbitrary implementations of an abstract type provide, while leveraging the ease of proving correctness that simpler implementations offer.

2.1 Introduction by Example

Continuing our running example, consider an arbitrary implementation of the `LifoStack<T>` abstract type: `ArbitraryLifoStack<T>`. Suppose we equip `ArbitraryLifoStack<T>` with an additional member:

```
convert: ArbitraryLifoStack<T> -> NilConsLifoStack<T>
```

Figure 4: An extra member for an arbitrary implementation of `LifoStack<T>`

Informally, convert maps an ArbitraryLifoStack<T> to its NilConsLifoStack<T> equivalent. Applying an ArbitraryLifoStack.push operation to an ArbitraryLifoStack<T>, and then applying convert should be equivalent to first applying convert, and then applying an equivalent NilConsLifoStack.push operation to the resulting NilConsLifoStack<T>; if this holds, we will say that ArbitraryLifoStack.push and NilConsLifoStack.push commute over convert. The same should be true of every other LifoStack<T> provided value:

```
A.convert A.empty = N.empty

(* forall l, x *)
A.convert (A.push l x) = N.push (A.convert l) x

begin
  match A.pop l with
  | None -> None
  | Some (x, l') -> Some (x, A.convert l')
end = N.pop (A.convert l)

A.size l = N.size (A.convert l)
```

Figure 5: Commutativity properties for the LifoStack<T> abstract type. For brevity, ArbitraryLifoStack is abbreviated as A, and NilConsLifoStack is abbreviated as N. I will continue to use these abbreviations for the remainder of the paper.

If each of ArbitraryLifoStack<T>'s and NilConsLifoStack<T>'s provided values commute over convert, we will say that ArbitraryLifoStack<T> and NilConsLifoStack<T> commute. If ArbitraryLifoStack<T> and NilConsLifoStack<T> commute, then any property that holds for NilConsLifoStack<T> also holds for ArbitraryLifoStack<T>, *up to* A.convert. I will describe what this means through an example.

Consider the property pop (push x l) = **Some** (x, l). If it holds for NilConsLifoStack<T>, and ArbitraryLifoStack<T> commutes with NilConsLifoStack<T>, it can be shown that A.pop (A.push x l) = **Some** (x, l') where A.convert l = A.convert l'. This is similar to, but slightly weaker than, the original property we had about NilConsLifoStack<T>; we previously required l = l'[1], and now we only require convert l = convert l'.

In general, as a property transfers from the canonical implementation to a commuting implementation, two elements of the abstract type that were previously equal may become only equal under convert. This is usually not an issue, since it only applies to expressions of the abstract type itself, and not to other values. In the case of LifoStack<T>, the resulting

[1]One can rewrite pop (push x l) = **Some** (x, l) as pop (push x l) = **Some** (x, l') where l = l'.

stacks from `push` and `pop` operations are equivalent only up to `convert`, but the elements returned by `pop` and the sizes returned by `size` are fully equivalent. Clients of an abstract type only interact with the abstract type through its provided values, all of which commute over `convert`. Therefore, equivalence up to `convert` and full equivalence are indistinguishable to the user.

In the special case where `convert` is injective, then equivalence up to `convert` is identical to full equivalence.

To prove that a property that holds in a canonical implementation also holds in a commuting implementation (up to conversion), straightforward symbolic manipulation almost always suffices. However, to gain intuition as to why any property can be "transferred" from a canonical implementation to a commuting implementation, it is helpful to view the property as a commutative diagram. If the property holds for the canonical implementation, then this diagram will consist of commutative subdiagrams, and thus commute.

$$\begin{tikzcd}
\texttt{unit} \arrow[rrr, "\texttt{1}"] \arrow[dr, "\texttt{A.empty}"] \arrow[ddr, "\texttt{N.empty}"'] & & & \texttt{int} \\
& \texttt{NilCons} \arrow[r, "\texttt{N.push \_ x}"] & \texttt{NilCons} \arrow[ur, "\texttt{N.size}"] & \\
& \texttt{Arbitrary} \arrow[u, "\texttt{A.convert}"] \arrow[r, "\texttt{A.push \_ x}"] & \texttt{Arbitrary} \arrow[u, "\texttt{A.convert}"'] \arrow[uur, "\texttt{A.size}"'] &
\end{tikzcd}$$

Figure 6: A commutative diagram demonstrating that `A.size (A.push x A.empty) = 1`, given that `N.size (N.push x N.empty) = 1` and that `A` and `N` commute. For simplicity, `1`, `N.empty`, and `A.empty` are treated as functions taking a `unit` argument.

This transference of correctness properties from the canonical implementation to arbitrary commuting implementations is valuable for two reasons. First, it allows us to extract properties that complicated implementations satisfy by only proving these properties for simple implementations. Second, it provides flexibility in the face of unsound correctness properties. Suppose an abstract type's collection of correctness properties is found to be unsound, and we revise the collection so that it becomes sound. Normally, we would need to prove the properties in the new collection for every implementation to ensure that every implementation is correct. With commutativity, we only need to prove this new collection of properties for the simple implementation, and these properties will transfer to every other implementation "for free".

2.2 Formalizing Commutativity

We will now formalize what it means for expressions to commute. Let `Ic<T1, ..., Tk>` and `Ia<T1, ..., Tk>` be canonical and arbitrary implementations of an abstract type, respectively. Let `convert : Ia<T1, ..., Tk> -> Ic<T1, ..., Tk>` be a conversion function from the arbitrary implementation to the canonical implementation. Two expressions `e'` and `e` commute over `convert`, denoted `e'` $\backsimeq$ `e`, according to the following rules:

$$\frac{\texttt{convert e' = e}}{\texttt{e'} : \texttt{Ia<}\tau_1, \ldots, \tau_k\texttt{>} \backsimeq \texttt{e} : \texttt{Ic<}\tau_1, \ldots, \tau_k\texttt{>}} \qquad \frac{\texttt{e' = e} \quad \tau \text{ does not contain } \texttt{Ia} \text{ or } \texttt{Ic}}{\texttt{e'} : \tau \backsimeq \texttt{e} : \tau}$$

$$\textsc{Product}\ \frac{\texttt{e' = (e1', e2')} \quad \texttt{e = (e1, e2)} \quad \texttt{e1'} : \tau_1' \backsimeq \texttt{e1} : \tau_1 \quad \texttt{e2'} : \tau_2' \backsimeq \texttt{e2} : \tau_2}{\texttt{e'} : \tau_1' \times \tau_2' \backsimeq \texttt{e} : \tau_1 \times \tau_2}$$

$$\textsc{Function}\ \frac{\forall\ \texttt{e' e},\ \texttt{e'} : \tau_1' \backsimeq \texttt{e} : \tau_1 \implies \texttt{f' e'} : \tau_2' \backsimeq \texttt{f e} : \tau_2}{\texttt{f'} : \tau_1' \to \tau_2' \backsimeq \texttt{f} : \tau_1 \to \tau_2}$$

$$\textsc{TaggedSum}\ \frac{\texttt{e' = Ci ei'} \quad \texttt{e = Ci ei} \quad \texttt{ei'} : \tau_i' \backsimeq \texttt{e\_i} : \tau_i}{\texttt{e'} : \texttt{C1 of } \tau_1' \texttt{ |...| Ck of } \tau_k' \backsimeq \texttt{e} : \texttt{C1 of } \tau_1 \texttt{ |...| Ck of } \tau_k}$$

Figure 7: Inference rules for expression commutativity, with product, function, and tagged sum types. Commutativity can also be extended to other typing constructs.

Arbitrary implementation `Ia` and canonical implementation `Ic` commute if and only if for each provided value `v`, `Ia.v` $\backsimeq$ `Ic.v`. As an example, let's look at what it means for `ArbitraryLifoStack.push` and `NilConsLifoStack.push` to commute:

$$\frac{\forall\ \texttt{l' l},\quad \dfrac{\texttt{A.convert l' = l}}{\texttt{l'} : \texttt{A<T>} \backsimeq \texttt{l} : \texttt{N<T>}} \implies \dfrac{\ddots\ \textit{(see below)}\ \iddots}{\texttt{A.push l'} : \texttt{T} \to \texttt{A<T>} \backsimeq \texttt{N.push l} : \texttt{T} \to \texttt{N<T>}}}{\texttt{A.push} : \texttt{A<T>} \to \texttt{T} \to \texttt{A<T>} \backsimeq \texttt{N.push} : \texttt{N<T>} \to \texttt{T} \to \texttt{N<T>}}$$

$$\frac{\forall\ \texttt{x' x},\quad \dfrac{\texttt{x' = x}}{\texttt{x'} : \texttt{T} \backsimeq \texttt{x} : \texttt{T}} \implies \dfrac{\texttt{A.convert (A.push l' x') = N.push l x}}{\texttt{A.push l' x'} : \texttt{A<T>} \backsimeq \texttt{N.push l x} : \texttt{N<T>}}}{\texttt{A.push l'} : \texttt{T} \to \texttt{A<T>} \backsimeq \texttt{N.push l} : \texttt{T} \to \texttt{N<T>}}$$

Figure 8: Formal derivation of `A.push` and `N.push` commuting

Putting this together gives us:

```
  ∀ l' l x' x, A.convert l' = l ⟹
                 x' = x ⟹ A.convert (A.push l' x') = N.push l x
≡ ∀ l x, A.convert (A.push l x) = N.push (A.convert l) x
```

which is exactly the commutativity property listed above.

3 Instantiations

To explore employing the commutativity paradigm in practice, I instantiated it in two abstract types using the Coq proof assistant: one abstract type for natural numbers, and another abstract type for arrays. All of the code can be found in a public Github repository (link here).

3.1 Instance 1: Natural Numbers

The standard implementation of the natural numbers in Coq is based on a unary representation. It is easy to prove properties about the natural numbers with this simple and elegant representation. However, the obvious downside of this implementation is its glaring inefficiency. In response to this inefficiency, more efficient implementations of the natural numbers have been added to Coq. In particular, the `BinNums` library offers a binary implementation of the natural numbers.

Any program that uses the natural numbers should not depend on how the natural numbers are implemented; instead, the program should only rely on the existence of particular values, such as zero, successor and addition, and properties that these values satisfy. This means that users of natural numbers could use an abstract type of natural numbers, as opposed to one particular implementation.

I wrote an abstract type for natural numbers using Coq's module system, and wrote multiple implementations of this abstract type using the commutativity paradigm:

- `NatInterface`: a module type which lists the values that a natural numbers implementation must provide
- `CanonicalNatImpl`: an implementation of `NatInterface` using the standard unary representation
- `CommutingNatInterface`: a module type extending `NatInterface` which introduces a `convert` value. It requires every provided value to commute with the corresponding `CanonicalNatImpl` value over `convert`. It also requires `convert` to be injective, which makes future proofs much cleaner.

- `CommutingCanonicalNatImpl`: an implementation of `CommutingNatInterface` using `CanonicalNatImpl`. Its `convert` value is the identity function, and all commutativity properties hold trivially.

- `VerifiedNatInterface`: a module type extending `CommutingNatInterface`, which includes a (possibly non-exhaustive) list of properties that a correct implementation of the natural numbers should satisfy.

- `VerifiedCommutingNatImpl`: a functor which accepts as an argument an implementation of `CommutingNatInterface`, and produces a corresponding implementation of `VerifiedNatInterface`

- `BinaryNatImpl`: an implementation of `CommutatingNatInterface` that uses `BinNums`'s binary representation.

In particular, the `VerifiedCommutingNatImpl` functor proves correctness properties for an arbitrary implementation of `CommutingNatImpl` using only the definitions in `CanonicalNatImpl` and proof that the commuting implementation does in fact commute.

3.2 Instance 2: Naturally-Indexed Arrays

In addition to developing commutative implementations of abstract types, I also wanted instantiate a client's use of commutative implementations. To achieve this, I created an abstract type for arrays indexed by the natural numbers. Instead of relying upon a particular implementation of the natural numbers, the interface takes as an argument an implementation of `VerifiedNatInterface` to use as indices:

- `ArrayInterface`: a module type listing the values that an array implementation must provide. Takes an implementation of `VerifiedNatInterface` as an argument, which is used for the arrays' indices.

- `CanonicalArrayImpl`: an implementation of `ArrayInterface` using nil-cons lists.

- `CommutingArrayInterface`: a module type extending `ArrayInterface` which introduces a `convert` value. It requires every provided value to commute with the corresponding `CanonicalArrayImpl` value over `convert`.

- `CommutingCanonicalArrayImpl`: an implementation of `CommutingArrayInterface` using `CanonicalArrayImpl`. Its `convert` value is the identity function, and all of the commutativity properties hold trivially.

- `VerifiedArrayInterace`: an extension of `CommutingArrayInterface` which includes a (non-exhaustive) list of properties that a correct array implementation should satisfy.

- `VerifiedCommutingArrayImpl`: a functor which accept as an argument an implementation of `CommutatingArrayInterface`, and produces a corresponding implementation of `VerifiedArrayInterface`.

- `TreeArrayImpl`: an implementation of `CommutatingArrayInterface` using binary trees.

Similar to `VerifiedCommutingNatImpl`, the `VerifiedCommutingArrayImpl` functor proves correctness properties for an arbitrary implementation of `CommutingArrayInterface` using only the definitions in `CanonicalArrayImpl`, and proof that the commuting implementation does in fact commute.

4 Conclusion

With all of this said, how do commuting implementations of abstract types build on top of what we've covered in CS 4860? The main ways in which this project relates to the course are its use of dependent types to formulate commutativity properties in first-order-logic, its use of refinement logic to prove these properties, and its use of constructive type theory as a foundation of computation.

The commutativity properties for abstract type implementations are stated using first order logic. When these properties are expressed as types, via the Curry-Howard isomorphism, they are dependent types. In the Coq instantiations of the commutativity paradigm, commutativity properties are expressed as dependently-typed values that commuting implementations must provide.

Additionally, I used refinement logic to prove these commutativity properties, and to prove that commutativity properties in turn imply more general properties. The refinement logics supported by proof assistants like Coq and NuPRL are built upon the refinement logic for first order logic that we covered in lecture.

Finally, the commutativity paradigm, and its instantiations in Coq, are founded upon constructive type theory. In particular, the distinction between canonical and non-canonical values that is made in constructive type theory shows up in commutativity properties. Every commutativity property contains subterms of the form `e1 = e2`, where `e1` and `e2` are expressions. The proposition `e1 = e2` is not asserting that `e1` and `e2` are syntactically equivalent, but instead that they reduce to the same canonical value.

In addition to pertaining to the material we covered in CS 4860, commuting implementations also are of practical interest. The most immediate application of logic to computer science is software verification. Proof assistants and other advances in formal methods have enabled the verification of increasingly sophisticated software, from kernels to compilers. Meanwhile, software systems have come to underlie finance, defense, medicine, and many of the world's most critical institutions. These increasingly high-stakes roles create a pressing demand for error-free systems.

Unfortunately, proving software correct remains an expensive endeavor, and for this reason the vast majority of software has not been formally verified. Finding less expensive methods for verification that still provide formal guarantees will enable wider adoption of software verification. The commutativity paradigm is a step in this direction, as it allows correctness properties proven for a simple implementation of an abstract type to be extended to arbitrarily complex implementations at no additional cost. It is my hope that this work becomes a small step towards making the world's software more reliable.

5 Acknowledgments

Thanks to Professor Robert Constable for teaching the logic that underlies this work in CS 4860.

Thanks to Adam Chlipala and Michael Nahas for their tutorials of Coq, which helped me become familiar with the proof assistant.

Thanks to many unnamed StackOverflow contributors whose LaTeX advice made this report's formatting much more pleasant and digestible than it would have been otherwise.

References

[1] A. Chlipala, *Certified programming with dependent types: A pragmatic introduction to the Coq proof assistant*. Cambridge, MA: The MIT Press, 2014.

[2] R. Constable, "CS 4860: Applied Logic," 2016.

[3] M. Nahas, "A tutorial by Mike Nahas," in *The Coq Proof Assistant*, 2014. [Online]. Available: https://coq.inria.fr/tutorial-nahas. Accessed: 2016.

[4] R. M. Smullyan, *First-order logic*. New York: Dover Publications, 2003.