Appendices

Contents

In addition to these appendices, we supply a full distribution of our source code, an annotated collection of microbenchmarks, and a technical report on the Facade language as separate downloads.

A A diagram of the compilation pipeline

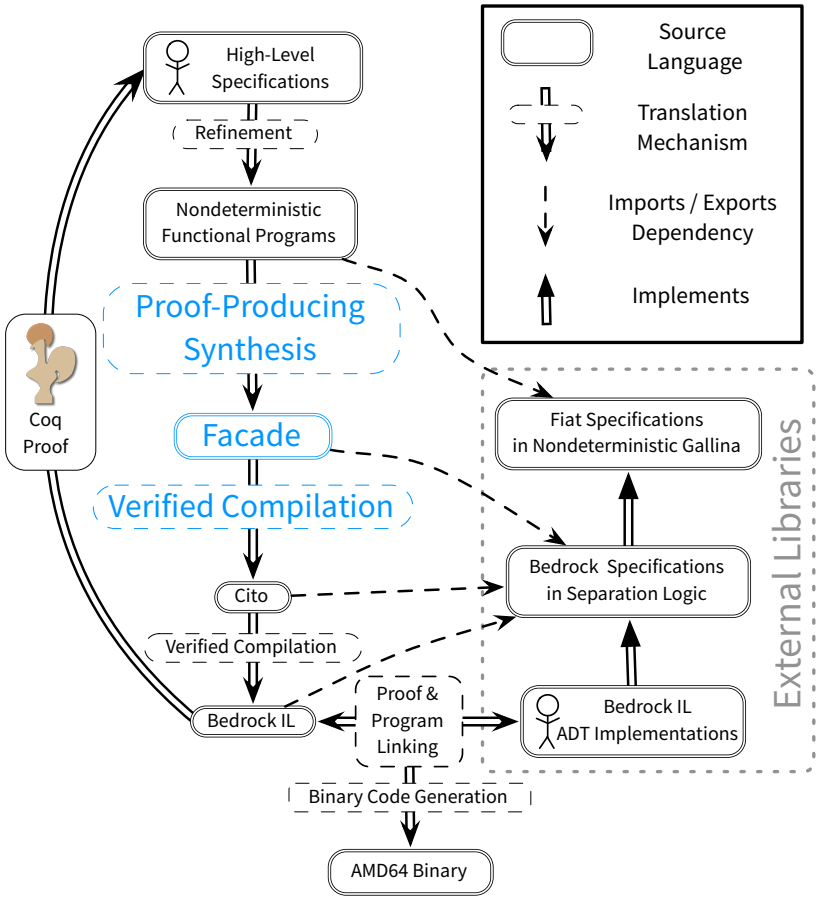

Figure 1: The full compilation pipeline, with the contributions of this work highlighted in blue. Stick figures indicate application-specific components supplied by the programmer.

B Other compilation rules: If, Bind, and Chomp

$$\frac{p_{Test} \Longrightarrow \varnothing \overset{\mathtt{p_{True}}}{\underset{[cmp \mapsto p_{Test}] :: ext}{\rightsquigarrow}} \mathtt{k} \mapsto \underline{\mathtt{p_{True}}} \qquad \neg p_{Test} \Longrightarrow \varnothing \overset{\mathtt{p_{False}}}{\underset{[cmp \mapsto p_{Test}] :: ext}{\rightsquigarrow}} \mathtt{k} \mapsto \underline{\mathtt{p_{False}}}}{\begin{array}{l}\mathtt{p_{Test}}\ \mathtt{;}\\ \mathtt{If\ cmp\ Then\ p_{True}}\\ \mathtt{Else\ p_{False}\ \ EndIf}\end{array}\quad \varnothing \overset{\mathtt{p}}{\underset{ext}{\rightsquigarrow}} [\![\mathtt{k} \mapsto \mathtt{if}\ p_{Test}\ \mathtt{then}\ \underline{\mathtt{p_{True}}}\ \mathtt{else}\ \underline{\mathtt{p_{False}}}]\!]}\ \textsc{If}$$

(a) The *if* rule: provided that the three intermediate programs `pTest`, `pTrue`, `pFalse` respectively evaluate the condition of the if and store the result in `cmp`, implement its left branch, and implement its right branch, we can connect the semantics of Facade's `If` statement to the Gallina-level *if*.

$$\frac{\forall\ v_0.\ v_0 \in \underline{\mathtt{v}} \Longrightarrow t\ v_0 \overset{\mathtt{p}}{\underset{[k \mapsto v_0] :: ext}{\rightsquigarrow}} t'\ v_0}{[\![\mathtt{k} \mapsto \underline{\mathtt{v}}\ \mathtt{as}\ v_0]\!] :: t\ v_0 \overset{\mathtt{p}}{\underset{ext}{\rightsquigarrow}} [\![\mathtt{k} \mapsto \underline{\mathtt{v}}\ \mathtt{as}\ v_0]\!] :: t'\ v_0}\ \textsc{Chomp}$$

(b) The *chomp* rule: to synthesize a program whose pre- and postconditions share the same prefix $[\![\mathtt{k} \mapsto \underline{\mathtt{v}}]\!]$, it is enough to synthesize a program that works for any constant values permitted by the Fiat computation $\underline{\mathtt{v}}$.

$$\frac{st \overset{\mathtt{p}}{\underset{ext}{\rightsquigarrow}} [\![\_ \mapsto \underline{\mathtt{comp}}\ \mathtt{as}\ x]\!] :: [\![\mathtt{k} \mapsto \underline{\mathtt{f}}\ x]\!] :: st'}{st \overset{\mathtt{p}}{\underset{ext}{\rightsquigarrow}} \left[\!\!\left[\mathtt{k} \mapsto \begin{array}{l} x \leftarrow \underline{\mathtt{comp}} \\ \underline{\mathtt{f}}\ x \end{array}\right]\!\!\right] :: st'}\ \textsc{Bind}$$

(c) The *bind* rule: dependencies between consecutive bindings in Fiat states accurately model the semantics of Fiat's `bind` operation.

C More on the $\mathtt{st} \lesssim st \uplus ext$ relation

The $\mathtt{st} \lesssim st \uplus ext$ is crucial to define the semantics of our Hoare triples. Here are some valid examples of the relation:

```
                  ["x" ↦ 1] :: ["y" ↦ (1,1)] ≲ ⟦"y" ↦ return (1,1)⟧ ∪ ["x" ↦ 1]
                  ["x" ↦ 1] :: ["y" ↦ (1,1)] ≲ ⟦"x" ↦ return 1⟧ :: ⟦"y" ↦ return (1,1)⟧ ∪ ∅
                  ["x" ↦ 1] :: ["y" ↦ (1,1)] ≲ ⟦"x" ↦ any as x⟧ :: ⟦"y" ↦ return (x,x)⟧ ∪ ∅
    ["x" ↦ 1] :: ["y" ↦ (1,1)] :: ["z" ↦ 2] ≲ ⟦"x" ↦ any as x⟧ :: ⟦"y" ↦ return (x,x)⟧ ∪ ∅
```

Note how, in the last example, we lost track of the `["z" ↦ 2]` binding; since 2 is a scalar, that is not an issue.

In contrast, here are some invalid examples:

```
                                  ["x" ↦ 1] ≴ ⟦"y" ↦ return (1,1)⟧ ∪ ["x" ↦ 1]
                     ["x" ↦ 1] :: ["y" ↦ (1,2)] ≴ ⟦"x" ↦ any as x⟧ :: ⟦"y" ↦ return (x,x)⟧ ∪ ∅
["x" ↦ 1] :: ["y" ↦ (1,1)] :: ["z" ↦ (1,2)] ≴ ⟦"x" ↦ any as x⟧ :: ⟦"y" ↦ return (x,x)⟧ ∪ ∅
```

In the first example, `"y"` is missing; this is an issue because `(1,1)` is an instance of the pair ADT, not a scalar. In the second example, the value of `"y"` is not consistent with `(x,x)`. In the last example, we lost track of the `["z" ↦ (1,2)]` binding; since `(1,2)` is encoded as an ADT value, that is an issue.

The Coq definition of this relation, matching the deduction rules given in Figure 4 of the paper, is presented below:

```
Fixpoint SameValues {av} ext fmap (tenv: Telescope av) ≜
  match tenv with
  | Nil ⇒ WeakEq ext fmap
  | Cons T key val tail ⇒
    match key with
    | NTSome k _ ⇒
      match StringMap.find k fmap with
      | Some v ⇒ ∃ w, w ∈ val ∧ v = wrap w ∧
                      SameValues ext (StringMap.remove k fmap) (tail w)
      | None ⇒ ⊥
      end
    | NTNone ⇒ ∃ w, w ∈ val ∧ SameValues ext fmap (tail w)
    end
  end.
```

D Case studies in extending the compiler

D.1 Intrinsics

A convenient feature of optimizing compilers is intrinsics, giving direct access in the source language to low-level primitives. To replicate this pattern in our extraction engine, we added a rule to the compiler that matches each application of a Gallina function `f` against a collection of user-supplied external Facade procedures to find one whose input-output specification matches `f` exactly. Should such a function be found, the compiler simply emits a call to the external procedure (Fig. 2).

```
Definition nibble_pow2 (w:W32) ≜ List.Inb w [1;2;4;8].

A Fiat program using the Gallina function nibble_pow2:
Example micro_nibble_power_of_two:
  ParametricExtraction
    #vars      x
    #program   ret (nibble_pow2 (x + 1))
    #arguments ⟦"x" ↦ ret x⟧
Proof. compile. Defined.

Without intrinsics, it compiles to multiple nested conditionals:
y = x + 1; If y == 1 Then out = 1
           ElseIf y == 2 Then out = 1
           ElseIf y == 4 Then out = 1
           ElseIf y == 8 Then out = 1
           Else out = 0 EndIf

With intrinsics, it compiles to a single, efficient call:
arg = x + 1; out = Call Intrinsics.nibble_pow2(arg);
```

Figure 2: Support for intrinsics allows us, for example, to efficiently extract snippets doing bit manipulations. Instead of unfolding its definition, we compile the Fiat program above by calling out to a low-level primitive; enabling this optimization is a one-line change (see '`micro_nibble_power_of_two`' in the anonymized '`MicrobenchmarksAnnotated.v`' file).

D.2 Rewritings

In many cases, implementing support for a new low-level primitive is not required to compile a new pattern; instead, we can transform the source program to re-express it in terms of constructs that we know how to handle.

For example, the default compilation tactic for microbenchmarks knows how to handle `revmap` (`revmap` is defined thus: `revmap f s ≜ map f (rev s)`) but not `map` (`revmap` is a special case of a left fold, in which accumulation is just consing to the head of an output list). Starting from the following example, compilation stops on the goal below:

```
ParametricExtraction
  #vars      seq
  #program   ret (map (λ w ⇒ w × 2) seq)
  #arguments ⟦ ‘"seq" ↦ seq ⟧ :: Nil
  #env       env.
```

$$⟦\texttt{"seq"} \mapsto \texttt{ret seq}⟧ \overset{?p}{\underset{\varnothing}{\rightsquigarrow}} ⟦\texttt{"out"} \mapsto \texttt{map (λ w ⇒ w × 2) seq}⟧$$

We could prove a separate lemma about `map`, but it is just as simple to implement it as a rewrite:

```
Lemma map_as_revmap {A B} :
  ∀ ls (f: A → B), map f ls = revmap f (rev ls).
Proof.
  unfold revmap; setoid_rewrite rev_involutive; reflexivity.
Qed.
```

We can then prove a similar lemma for `rev`; once we extend the compiler with these two extra facts, compilation completes effortlessly:

```
seq’ = List[W32].nil()
test = List[W32].empty?(seq)
While (not test)
    head = List[W32].pop(seq)
    Call List[W32].push(seq’, head)
    test = List[W32].empty?(seq)
EndWhile
Call List[W32].delete(seq)
out = List[W32].nil()
test = List[W32].empty?(seq’)
While (not test)
    head = List[W32].pop(seq’)
    r = 2
    head’ = head * r
    Call List[W32].push(out, head’)
    test = List[W32].empty?(seq’)
EndWhile
Call List[W32].delete(seq’)
```

E A diagram of Theorem 1

The following diagram summarizes Theorem 1 in graphical form.

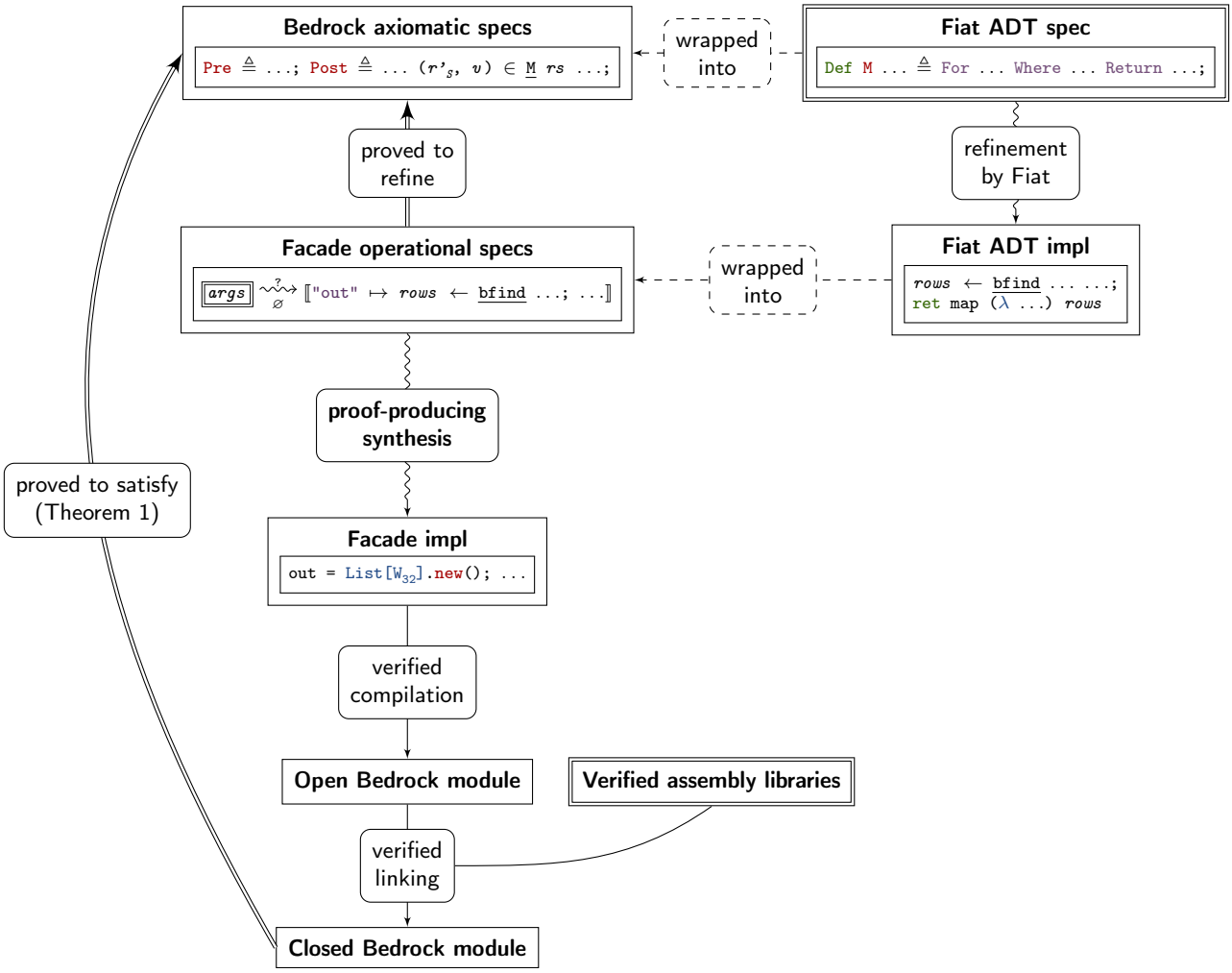

Figure 3: Visual summary of Theorem 1: Wavy arrows represent proof-producing transformations, solid arrows are verified code transformations, and double arrows are generic proofs connecting various parts of the pipeline. Double-borders indicate user-supplied sections.

F Connecting Fiat and Bedrock specifications: the FindFirst call rule

The following figure shows a detailed example of how complex call rules are elaborated Fiat and Bedrock specifications are connected: a Bag operation specified in Fiat is translated into a binary tree operation at the Bedrock level, through a *call rule* whose correctness is guaranteed by a machine-checked proof. Interestingly, while our extraction strategy has a strong flavor of synthesis, proving call rules correct has a strong flavor of traditional imperative program verification, where Fiat telescopes serve as a specification language for known Facade programs, instead of diving the derivation of an unknown Fiat program.

fiat/src/QueryStructure/Implementation/DataStructures/BagADT/BagADT.v

```
Def bfind (r: rep) (k: SearchTermType): rep * (list ElementType) ≜
  results ← {seq | ListEnsembleEquiv seq
                   (r ∩ (λ x ⇒ MatchSearchTerm k x = true))};
  return (r, results)
```

(a) Fiat specification of the bfind method, as written by the original Fiat authors. Running bfind on a bag `r` must return the bag unmodified, plus a list of all elements of `r` matching the search term `k`. This very general interface is known by the Fiat synthesizer, and allows it to soundly issue calls to the bag ADT.

bedrock/Bedrock/Platform/Facade/examples/QsADTs.v

```
Definition findFirst: AxiomaticSpec ≜ {|
  Pre ≜ λ args ⇒
          ∃ db key, args = [ADT (Tree db);
                            Scalar key]
                        ∧ functionalIndices db;
  Post ≜ λ args retv ⇒
           ∃ db key l, args = [(ADT (Tree db), Some (Tree db));
                               (Scalar key, None)]
                          ∧ ListEnsembleEquiv l (keepEq db 1 key)
                          ∧ retv = TupleList l |}.
```

(b) Bedrock specification of the `BTree.FindFirst` method, which implements a version of the bfind method specialized to nested trees, applicable when the argument of bfind describes an operation implementable using the first level of nested trees. `BTree.FindFirst` specifies a function of two arguments (a nested tree `db` and a scalar `k`) that returns a list `l` of all tuples of `db` whose first indexed field matches `k`, without observably modifying the underlying data structure.

$$\frac{\begin{array}{cc} ext\,[\mathtt{x_{Key}}] = key & \Psi[\mathtt{fp}] = \mathtt{BTree.FindFirst} \\ \mathtt{x\_Success}, \mathtt{x_{Table}} \notin ext & \mathtt{tupleSize(db)} < 2^{32} \\ \mathtt{x_{Table}} \neq \mathtt{x_{Key}} \neq \mathtt{x_{Success}} & \mathtt{functionalIndices(db)} \end{array}}{\llbracket \mathtt{x_{Table}} \mapsto \mathtt{return}\ db \rrbracket :: st \overset{\substack{\mathtt{x_{Success}} = \\ \mathtt{Call\ fp(x_{Table},\ x_{Key})}}}{\underset{ext}{\rightsquigarrow}} \begin{array}{l} \llbracket \underline{\mathtt{bfind}}\ db\ (key, \_, \_)\ \mathtt{as}\ (db', b) \rrbracket :: \\ \llbracket \mathtt{x_{Table}} \mapsto \mathtt{return}\ db' \rrbracket :: \\ \llbracket \mathtt{x_{Success}} \mapsto \mathtt{return}\ b \rrbracket :: st \end{array}}\ \textsc{CallBFind}$$

(c) One of the call rules for bfind, connecting the two interfaces above. The antecedents ensure that `key` is available as $\mathtt{x_{Key}}$, that there are no name collisions, that `fp` is a pointer to a function with the right Bedrock specification, that the length of a tuple fits a single machine word, and that `db` does not have duplicate indices. The consequent states that emitting a Facade-level `Call` instruction is enough to implement a call to bfind at the Fiat level.

Figure 4: Connecting Fiat and Bedrock specifications.

G Trusted base and compiler performance

Our derivation is assumption-free in the Calculus of Inductive Constructions extended with Ensemble Extensionality and Axiom K. Our trusted base comprises the Coq proof checker (∼10,000 lines of OCaml code), the semantics of our assembly language (∼1000 lines of Gallina specifications), the translator from Bedrock IL to x86 assembly (∼200 lines of Coq code), an assembler, and hand-written wrappers for the extracted methods (∼50 lines of x86 assembly). Details about our implementation and proof styles are given below.

On the SQL example, the extraction process, bottlenecked by inefficient symbolic manipulations in the core of Coq, takes about 9 minutes on an Intel Core i7-4810MQ @ 2.80GHz: 5 min. for synthesis and 4 min. for proof-checking.

H Implementation details and proof style

The core of our extraction framework consists of 5000 lines of specifications and theorem statements and 1700 lines of Coq proofs (checked in Coq v8.4pl2). Most of the derivations are done in a highly automated manner, with one main proof strategy per type of statement: a low-level tactic that systematically unfolds the $\lesssim$ relation; an intermediate one that that treats the $\lesssim$ relation abstractly (using only properties proved with the lower-level tactic) and always unfolds the $\rightsquigarrow$ relation; and one that treats $\rightsquigarrow$ abstractly.

This division of labor can be better understood by noticing that the $\rightsquigarrow$ relation has the pleasant property of being defined in high-level terms: it only manipulates Fiat states through the $\lesssim$ relation, without explicitly looking at the contents of these states. This makes $\rightsquigarrow$ a morphism for an equivalence relation $\equiv$ on Fiat states defined by connecting two Fiat states if they describe exactly the same Facade states (in terms of $\lesssim$). This makes the second category of lemmas easy to express, owing to the close parallel between Fiat computations and Fiat states, by reducing them to lemmas about $\equiv$. Concretely, this means that we can reduce proving the *bind* rule for $\rightsquigarrow$ (Figure 2c) to a proof that the right-hand sides of the antecedent and consequent are equivalent under $\equiv$. Concretely, this means that it is enough to prove

$$\left[\!\!\left[\mathtt{k} \mapsto \begin{array}{l} x \leftarrow \underline{\mathtt{comp}} \\ \underline{\mathtt{f}}\ x \end{array} \right]\!\!\right] :: st \equiv [\![\_ \mapsto \underline{\mathtt{comp}}\ \mathtt{as}\ x]\!] :: [\![\mathtt{k} \mapsto \underline{\mathtt{f}}\ x]\!] :: st$$

to derive

$$\frac{st \overset{\mathtt{p}}{\underset{ext}{\rightsquigarrow}} [\![\_ \mapsto \underline{\mathtt{comp}}\ \mathtt{as}\ x]\!] :: [\![\mathtt{k} \mapsto \underline{\mathtt{f}}\ x]\!] :: st'}{st \overset{\mathtt{p}}{\underset{ext}{\rightsquigarrow}} \left[\!\!\left[\mathtt{k} \mapsto \begin{array}{l} x \leftarrow \underline{\mathtt{comp}} \\ \underline{\mathtt{f}}\ x \end{array} \right]\!\!\right] :: st'} \textsc{Bind}$$

The proof style that we followed to verify the core parts of the framework consists in progressively constructing, as we elaborate the first proofs in a given class of lemmas (say, lemmas about $\lesssim$), an increasingly complete decision procedure for that class. That procedure is based on manually identifying and collecting repeatedly occurring patterns as new branches of a large Ltac match construct, thereby encoding rules to make progress from the various types of goals that we encounter. In a style favored by certain SMT solvers for dealing with quantifiers, we use a `learn` tactic that creates a specialized copy of a hypothesis and records

this specialization. This allows us to be liberal in the number of facts that we derive from the proof context, while avoiding repeated relearning of the same facts. In addition, this strategy suppresses much of the need for backtracking (we never lose the original hypothesis when learning a specialized one). This systematic approach yields relatively short proofs, with minimal duplication; and for particularly tricky lemmas, it does not preclude us from falling back to a more manual style to close certain branches.

The definition of `learn` is shown below. This tactic, given a proof a fact, checks whether this fact has already been learnt; if it has, it throws an exception; otherwise, it records the learnt fact, adds it to the context, and returns successfully. Original versions of this tactic checked for a hypothesis of the same type, but this check turned out to be too costly. In its current form, the `learn` tactic doesn't exactly check whether a given fact is already known before recording it; instead, it checks (syntactically) whether that fact has already been recorded using the `learn` tactic, failing quickly in the common case where the very same proof has already been used. In practice, this means that `learn` may create duplicates of already-known facts, and that `learn` will accept to record syntactically different but intensionally equal facts. Its implementation is very simple:

```
Inductive Learnt {A: Type} (a: A) :=
  | AlreadyKnown : Learnt a.

Ltac learn fact :=
  lazymatch goal with
  | [ H: Learnt fact ⊢ _ ] ⇒
    fail 0 "fact" fact "has already been learnt"
  | _ ⇒ let type := type of fact in
        lazymatch goal with
        | [ H: Learnt type _ ⊢ _ ] ⇒
          fail 0 "fact" fact "of type" type "was already learnt through" H
        | _ ⇒ pose proof (AlreadyKnown fact); pose proof fact
        end
  end.
```

I A different target domain: Binary encoders

The following `encode_MixedRecord` is an example of a binary encoder:

```
Record MixedRecord ≜
  { f1 : Byte; f2 : BoundedNat 8;
    f3 : BoundedList (BoundedNat 8) 2^8 }.

Definition EncodeMixedRecord (mr: MixedRecord) ≜
      (EncW8 0b00101010) Then (EncBNat mr.f2)
 Then (EncW8 mr.f1) Then (EncBNat (length mr.f3))
 Then (EncList EncBNat mr.f3).
```

A program in this binary-encoder DSL produces a list of bytes by combining encoders for base types, e.g. `encode_BdNat`, via the `Then` concatenation combinator. Our automated compilation strategy leverages the regular structure of these programs by applying a set of lemmas explaining how to convert each base encoder into an equivalent Facade function. The resulting Facade program stores the formatted data in a heap-allocated Bedrock ByteString ADT structure that can be passed directly to transmission code:

```
tmp = $128;
out = ByteString.new(tmp);
arg = 0b000101010;
Call ByteString.push!(out, arg);
arg = f1;
Call ByteString.push!(out, arg);
arg = f2;
Call ByteString.push!(out, arg);
arg = List[w32].length(f3);
Call ByteString.push!(out, arg);
test = List[w32].empty?(f3);
While (not test)
    head = List[w32].pop!(f3);
    arg  = head;
    Call ByteString.push!(out, arg);
    test = List[w32].empty(f3);
EndWhile
Call List[w32].delete!(f3)
```

While this code appears straightforward, our compilation strategy has enabled clever optimizations. First of all, while the original specification worked in terms of functional concatenation of lists, the extracted version repeatedly appends to an imperative buffer. Additionally, we have avoided compiling `encode_BdNat`, since the intermediate byte it would produce is equivalent to the original low-level representation of natural numbers, so instead we append that value directly onto the ByteString.

At this point, it is interesting to consider what this encoding program, extracted to OCaml, may look like:

- The source programs manipulate a dependently type "machine word" data type. Extracting this type naively would lead to bad performance and code bloat: in Fiat,

operations on this type are specified by roundtripping in and out of unbounded integers, and bit manipulations are done using recursive functions over the individual bits. Instead, when the number of bits in a given word matches one of the native integer types available in the target language, one could hope to extract directly to simple integer types. Unfortunately, the OCaml extraction engine is not able to extract the word type nonuniformly, so that e.g. `word 32` would be extracted to $\texttt{int}_{32}$, and `word 8` to `char`. This would force users of OCaml extraction to use a single type of large integers.

- The source programs produce output by explicitly reconstructing bytes from the data stored inside an input packet. For numbers, for example, this means using repeated division to extract individual bits and store them inside a value of type `word`. For strings, this means iterating over the characters of the string and then iterating over the bits of the representation of each character. This makes it easy to ensure that the source programs have proper semantics, but it has disappointing performance.
- The source programs produce sequences of bytes, with no particular focus on efficiency: they append to the end of a list using direct concatenation (concretely, to push a single byte out, they use `bytes ++ [new_byte]`). Again, this makes the sources easy to read as a description of the serialization protocol, but it yields bad performance if extracted directly. Instead, we would like to produce an uninterrupted chunk of machine memory as the output of the encoding process. Thus mirroring the exact semantics of the source program into OCaml is not ideal, both for performance and for code complexity (we would have to write additional, unverified code to convert the list of bytes produced by our compiled programs into an OCaml array).
- The source programs manipulate explicit records containing the fields of the packet being serialized; this means that if we want to expose our code to, say, a C library, that library will have to construct a C `struct` for us, which we will then read values from. It would arguably be better, performance-wise, to receive each of these fields as parameters to the encoding function (additionally, it would be best to be easily interoperable with such C code, instead of having to massage the C data to be able to pass it to our extracted encoders). Thus, it would be best for users to have a say on the low-level representation of the data manipulated by these binary encoders.

Fortunately, our approach addresses each of these issues:

- We don't need to extract the `word` data type uniformly; instead, we can soundly encode both bytes and 32-bit words into Facade scalars. Of course, using 32-bit scalars to represent 8-bit words is only correct as long as we don't do arithmetic on these words: the semantics of e.g. addition on 8-bit words are not the same as those on 32-bit words.
- If we choose the right machine representation for the data structures of the source programs, and if we choose the right machine-level representation of integers and strings, we can ensure that their in-memory layout will match the encoding used by the source program. Concretely, if the source program manipulates integers smaller than 2^8, we can choose to store them in memory as 8-bit Bedrock words. With this, the code that takes such a number, extracts each bit using repeated division, and concatenates these bits to a Bedrock word can be compiled into code that copies the input number: since it is already represented at the machine level in exactly the way we wish to encode it, the encoding operation is a no-op. Similarly, if we ensure that strings are represented as contiguous arrays of chars, then encoding a string becomes essentially a memory copy from one array of chars to another.

- We can recognize the pattern that consists in appending a single byte to a list of bytes, and choose as a wrapper for list of bytes a function that injects that list of bytes into a contiguous stack of bytes. Then appending maps to the push operation of these stacks, providing a sound, constant-time implementation of this otherwise linear-time operation.
- We are free to specify the form of the input to our encoders and decoders, and thus we can phrase the extraction problem so as to enforce that the program start with one argument variable per field in the original structure. Additionally, by carefully selecting the machine representation of each field, we can ensure easy interoperability with C code.