Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs

Clément Pit-Claudel Peng Wang Benjamin Delaware Jason Gross Adam Chlipala
MIT CSAIL
{cpitcla,wangpeng,bendy,jgross,adamc}@csail.mit.edu

Abstract

We present an original approach to sound program extraction in a proof assistant, using syntax-driven automation to derive correct-by-construction imperative programs from nondeterministic functional source code. Our approach does not require committing to a single inflexible compilation strategy and instead makes it straightforward to create domain-specific code translators. In addition to a small set of core definitions, our framework is a large, user-extensible collection of compilation rules each phrased to handle specific language constructs, code patterns, or data manipulations. By mixing and matching these pieces of logic, users can easily tailor extraction to their own domains and programs, getting maximum performance and ensuring correctness of the resulting assembly code.

Using this approach, we complete the first proof-generating pipeline that goes automatically from high-level specifications to assembly code. In our main case study, the original specifications are phrased to resemble SQL-style queries, while the final assembly code does manual memory management, calls out to foreign data structures and functions, and is suitable to deploy on resource-constrained platforms. The pipeline runs entirely within the Coq proof assistant, leading to final, linked assembly code inside Coq with overall full-functional-correctness proofs in separation logic.

1. Introduction

It is a holy grail of formal methods to make it routine to prove the functional correctness of every new program. Such a proof relies on someone to write a formal specification, capturing what "correctness" means in each particular case. Even more appealing is to have the program be generated automatically from the specification, producing a proof of correctness as a natural byproduct. Instead of doing the usual work of programming, followed by the probably even harder work of proving, this approach collapses the two into one process, which may even proceed more smoothly than either step in the other style.

The general area of correct-by-construction code generation is venerable, going back at least to Dijkstra's work in the 1960s [3]. Oftentimes, solutions offer a strict subset of the desiderata of generality, automation, and performance of synthesized code. This paper presents a pipeline that sits at the sweet spot of all three, enabling semiautomatic refinement of high-level specifications into efficient low-level code in a proof-generating manner. Our initial specification language is the rich, higher-order logic of Coq, and we support a high degree of automation through domain-specific refinement strategies, which in turn enable targeted optimization strategies for extracting efficient low-level code. In order to take advantage of these opportunities, we have built an *extensible* compilation framework that can be updated to *handle new compilation strategies without sacrificing soundness*.

One might expect a realistic derived program to combine textbook data structures and algorithms with problem-specific glue. It seems foolhardy to expect the derivation system to reinvent textbook ideas automatically; rather, the architecture should combine automation of the glue with a kind of foreign function interface to include manually verified ingredients. Proof-generating refinements should be integrated with *arbitrary pieces of assembly code that have been highly tuned for performance*, placing as few restrictions as possible on how that code was written (e.g., handwritten or compiled from some source language) or verified (e.g., manual proofs at the assembly level or proofs via a certified compiler).

We also aspire to a *foundational* tool, which produces a fully linked (closed) assembly program, represented as a formal object inside a proof assistant with a proof that it meets the original high-level specification. No part of our derivation process should be trusted, and neither should we trust the manually verified parts of the final program.

This paper presents a novel approach to certified extraction that achieves these three goals to be *extensible* and *correct*, all while producing *efficient* low-level code from high-level specifications. Our solution, realized in a prototype tool, enables mostly automated derivations of efficient assembly code from specifications by soundly supporting a division of labor where a tool finds the relatively straightforward parts of the program and humans supply crucial, domain-specific ingredients, compiled from arbitrary source languages and

manually verified with arbitrary proof methods.

At the high level, we begin with the Fiat program-derivation framework [2], which refines high-level specifications into programs written in Gallina (Coq's functional programming language), along with proofs that the resulting programs conform to the original specifications. A hallmark of the framework is its reliance on domain-specific automation of program derivations. As an example, Fiat's query-structures library automatically refines the following SQL-like query specification into a correct-by-construction program similar to the following one:

```
SELECT COUNT(*) FROM Books
JOIN Orders ON isbn
WHERE Books.by = $author
```

```
results ← bfind $books $author;
ret (fold_L (λ count book ⇒
               count + bcount $orders book.isbn)
            results 0)
```

In addition, Fiat programs can be parameterized by abstract data types (ADTs), with behavior axiomatically specified and implementations left undetermined. In the example above, bfind and bcount are methods of a bag ADT; the call to the former returns all books by a certain author `$author`, and the call to the latter counts the matches in the `$orders` table for a given ISBN code `.isbn`.

The code output by Fiat above is misleadingly innocuous. We do not, in fact, arrive at a normal, executable program in Gallina. The bfind method returns a list of database rows that match some condition, and it may return its results *in any order*. We do not grant this freedom spuriously. Imagine that we represent the database with a type of binary search tree that does not enjoy canonical representations. The same abstract data may wind up represented by multiple different candidate trees, and *each tree implies that a different order of* bfind *results is quickest to produce*. State-of-the-art relational-database engines take advantage of this same flexibility, picking result orders compatible with a database's chosen index structures. Therefore, to support code derivation with flexible use of efficient low-level data structures, *nondeterminism at runtime is essential*, from the perspective of the client code that interfaces with external data structures.

The original Fiat system resolves all nondeterminism in the context of functional programs in the nondeterminism monad, where $\mathsf{nondet}(\alpha)$ is the type of sets of α's, $\mathtt{ret}\ x \triangleq \{x\}$, and $\mathtt{bind}\ \underline{x}\ \underline{f} \triangleq \{y \in \underline{f}(v) \mid v \in \underline{x}\}$: specifications are highly nondeterministic programs, while final implementations are fully deterministic programs in the same monad. Every transformation to a program must be proved to *refine* its behavior, that is, move to some subset of the original set (possibly the exact same set). A final implementation of an operation is a singleton set, meaning that it commits to a single possible result. As a consequence, the code produced by the original Fiat system must commit to a particular implementation of the ADT operations that it relies on.

In previous work, the final step of a derivation was to *extract* this code directly to OCaml, using Coq's popular but unverified mechanism by that name. Unfortunately, this means that correctness of the compiled executable depends not only on the correctness of Coq's kernel but also on that of the extraction mechanism and of the OCaml compiler and runtime system. These two dependencies significantly decrease the confidence that users can place in programs synthesized by Fiat, and more generally in all programs extracted from Gallina code. Furthermore, with its reliance on garbage collection, OCaml is not tuned to squeeze maximum performance out of resource-constrained embedded platforms.

Our work overcomes these issues via what can be viewed as a proof-generating "extraction" process running within Coq, starting from nondeterministic Fiat programs and producing assembly code instead of OCaml code, done in a way that supports linking with handwritten and hand-verified assembly code. In contrast to the original Fiat system, which outputs fully deterministic programs, this paper allows Fiat's final output to incorporate nondeterminism (as demonstrated by the binding operation ($\leftarrow$) of our initial example), making it possible to resolve at a later stage by *interpreting the nondeterminism as a precondition-postcondition specification in Hoare logic* and *linking against arbitrary assembly code that has been proved to meet that specification*.

In contrast to well-known examples of certified compilation like CompCert [9] and CakeML [6], we do not implement our translator in the proof assistant's logic and prove it sound once and for all. The primary reason is that we want to make our compiler *extensible* by not committing to a specific compilation strategy. Programmers can instead teach it about new verified low-level data structures and code-generation strategies by registering as hints new lemmas explaining *how to implement a term in the nondeterminism monad using a particular low-level imperative program*. In fact, because Fiat programs can include arbitrarily complex nondeterministic choices, Fiat programs *cannot* be compiled systematically: compiling $x \leftarrow \{r \mid r = 0 \leftrightarrow (\exists\ a\ b\ c\ n,\ n \geq 3 \wedge a^n + b^n = c^n)\};\ \mathtt{ret}\ x$, for example, would require plugging a proof of Fermat's last theorem into our compiler. We introduce *a novel sequent-calculus-style formulation of this translation problem*, allowing us to apply all of Coq's usual scriptable proof automation to the problem of proof-generating code extraction.

Our automation repeatedly applies hint lemmas until the nondeterministic functional program is fully transformed into the (deeply embedded) syntax tree of an imperative program, with imports of methods that can be coded in other languages, with assumed Hoare specifications. At this point we rejoin the certified-compilation approach, using a verified compiler from this imperative language to Bedrock assembly [1]. We mostly reuse the existing verified Cito optimizing compiler [23], but we add on top a new source language *Facade* and a corresponding verified compiler;

Facade is a more natural target when translating from a functional language.

The many advantages of this approach (extensibility, ease of development, flexibility, and ease of verification) do come at a cost, however: compilation is slower, care is needed to make the compiler robust to small variations in input syntax, and the extensible nature of the compiler makes it hard to characterize the supported source language precisely. The basic comparison is much the same as between flexible higher-order logic and decidable first-order logics.

We combine all these pieces to build the complete certifying pipeline from high-level relational specifications to assembly presented in figure 1. Our toolchain uses Fiat [2] to refine high-level specifications of ADTs into nondeterministic functional programs (expressed in a shallowly embedded Gallina DSL) depending on external data structures, then soundly extracts these programs to Facade using a novel proof-generating compilation-by-synthesis procedure. The resulting programs are then translated to Bedrock's Cito language by a relatively simple desugaring, backed by a non-trivial soundness argument bridging two styles of operational semantics. The Bedrock pipeline does traditional verified compilation from the Cito level and soundly links against hand-verified implementations of the required data structures. Beyond exploring a new technique for sound extraction of shallowly embedded DSLs (EDSLs), this work bridges the last remaining gap to present the first mechanically certified automatic translation pipeline from declarative specifications to efficient assembly-language programs. We will have more to say about all of the pieces in the remainder of the paper, with section 4 presenting the details of the new pieces of the pipeline.

To summarize the benefits of our approach, as compared to other points in related design spaces:

- It is lightweight: it does not require reifying the entirety of Gallina into a deeply embedded language before compiling. Instead, we use Coq's Ltac language to drive compilation.
- It is extensible: each part of the compilation logic is expressed as a derivation rule, proved as an arbitrarily complex Coq theorem. Users can add support for new constructs and optimizations by supplying their own compilation lemmas extending the supported source language or improving the quality of the generated code.
- It is well suited to compiling EDSLs: we crucially support nondeterminism in input programs, where standard extraction requires deterministic code.
- It allows us to link against axiomatically specified libraries of assembly code, implemented and verified in arbitrary source languages and compiled to the Bedrock ecosystem.
- It compiles directly to a relatively bare language, with fully manual memory management.

To demonstrate the applicability of this approach, section 6 presents an evaluation of our extraction routines on benchmarks drawn from three different domains:

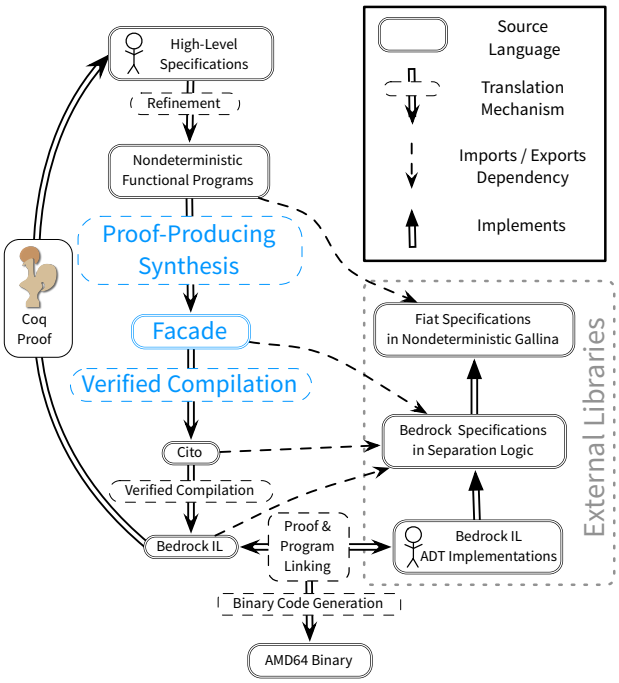

Figure 1: The full compilation pipeline from high-level specifications to assembly programs, with the contributions of this work highlighted in blue. Stick figures mark the application-specific components supplied by a programmer.

- A set of microbenchmarks of Fiat programs manipulating variables, conditions, and nested lists of machine words.
- Binary encoders that marshal in-memory data into binary formats suitable for transmission over a network.
- SQL-like programs similar to those of the original Fiat paper. These benchmarks start from high-level specifications of database queries and pass automatically through our pipeline to closed assembly programs, complete with full-functional-correctness specifications and proofs in separation logic.

Source code and compilation instructions for the framework and benchmarks are included as a supplement to this paper.

2. Outline of Approach

We begin with an example of the pipeline from figure 1 in action, starting with a relational specification of a query embedded in Coq in an SQL-style syntax similar to that provided by Fiat's query-structure domain. This particular query returns, in some order, the titles of all books written by a particular author, provided in the variable *$author*.

```
SELECT title FROM Books
WHERE Books.by = $author
```

Such a query would usually be one among many in a full specification. Fiat automatically decides which abstract underlying data structures would be useful for implementing the full set of operations, refining the specification into an efficient functional program using any data structure meeting

that interface. For this example, Fiat chooses dictionaries keyed on book authors, because we need to filter books by author. Using `IndexedByAuthor` to stand for some unknown implementation of that dictionary API, we wind up with a functional program like the following:

```
rows ← IndexedByAuthor.bfind $books $author;
ret (map (λ row ⇒ row.Title) rows)
```

This program, written in the nondeterminism monad, computes a list of rows for books with the right author and then maps over the rows to produce a list of only titles. Module `IndexedByAuthor` is not an executable implementation of the required functionality; rather, it specifies certain methods nondeterministically, saying that bfind returns the expected rows in some undetermined order. The order may even be different on every call, as might arise, for instance, with data structures like splay trees that adjust their physical structure even during logically "read-only" operations.

Such nondeterministic programs are the starting point for our new refinement phases. The ultimate output of the pipeline is a library of assembly code in the Bedrock framework [1]. Rather than generating assembly directly, our new phases produce code in the Cito C-like language [23], which is then compiled to Bedrock. The output for our running example might look like the following, where for clarity we use a more C-style syntax than Cito accepts literally.

```
TupleList *rows = BTree.find($books, $author);
TupleList *out  = new StringList();
while (!TupleList.empty(rows)) {
  Tuple *row   = TupleList.pop(rows);
  String title = Tuple.get(row, TITLE_COLUMN_ID);
  StringList.push(out, title); }
TupleList.delete(rows);
StringList.reverse(out);
return out;
```

Note that this code works directly with pointers to heap-allocated mutable objects, doing manual memory management on the objects we allocate, including for intermediate values. Also, the general `IndexedByAuthor` interface has been replaced with calls to a concrete library module `BTree`, providing binary search trees of tuples, keyed off of a particular column chosen at initialization time. For efficiency, we have compiled the use of `map` into an imperative loop. The verified compiler for Cito [23] translates a set of method bodies of this kind into a library of Bedrock assembly code. We implement and verify `BTree` and other referenced modules directly in other Bedrock-platform languages, and then we link code and proofs to produce a final library with no imported symbols. The theorem attached to this library (written in separation logic [15]) tells us that *each method behaves compatibly with its original formulation*. That is, we have full functional correctness of assembly libraries, for code generated automatically from high-level specifications.

The contribution of this paper is in spanning the gap from *the nondeterministic functional program* to *the imperative low-level program*, in an extensible and proof-generating way that coordinates calls to external functions that have nondeterministic specs. One (relatively simple) element of our approach is that we first translate to an intermediate language, called Facade, that is slightly higher-level than Cito. Facade no longer exposes a mutable heap, instead putting all mutable state in local variables (potentially shared across functions when passed by reference). Facade inherits the two least usual parts of Cito's operational semantics:

- The semantics is parameterized over an *abstract data type*, for instance mathematical sets to represent finite sets. Local variables appear to contain these abstract values directly; for instance, one machine state might say "variable x contains the mathematical set $\{2,4,6\}$." (In Cito, such values are associated with heap addresses instead of local variables.)
- Another parameter of the semantics is a separation-logic specification for each imported foreign function, where the states of mutable objects are described in terms of abstract values. Since Facade avoids using the heap, we no longer need separation logic and can merely refer to the (functional) values of variables, though we must introduce the notion of by-reference parameters, which allow controlled modification of the caller's variables by the callee. (The underlying Cito code just passes pointers stored in local variables, as Cito does not itself support by-reference parameters or pointers to local variables.)

A trivial verified compiler converts between the tiny syntactic differences in the two languages. Facade features a more restrictive syntax than Cito's, for instance forbidding function actual parameters to be variables. In this sense, Facade is more like a new type system for Cito than a new language. The Facade version of the Cito code shown above looks almost the same, with some uses of pointers replaced with by-reference parameter passing, but the semantics underlying the two forms are quite different, with important consequences for ease of verification.

The heart of our contribution is an extensible, proof-generating framework for deriving such Facade programs from functional programs in the nondeterminism monad. We phrase this derivation problem as one of finding a proof of a Facade Hoare triple, where the precondition and postcondition are known, but the Facade program itself must be derived during the proof. The central goal from our running example looks as follows, where `?1` stands for the overall Facade program that we seek, and where we unfold IndexedByAuthor.bfind to its (nondeterministic) definition.

$$[\![\texttt{"books"} \mapsto \texttt{ret } \$\textit{books}]\!] :: [\![\texttt{"author"} \mapsto \texttt{ret } \$\textit{author}]\!] \overset{?1}{\rightsquigarrow}_{\varnothing}$$
$$\left[\!\!\left[\texttt{"out"} \mapsto \begin{array}{l} r \leftarrow \text{shuffle}(\$\textit{books} \cap \{b \mid b.\texttt{by} = \$\textit{author}\}); \\ \texttt{ret } (\texttt{map } (\lambda\ \textit{row} \Rightarrow \textit{row}.\texttt{Title})\ r) \end{array}\right]\!\!\right]$$

The actual implementation of `?1` is found by applying lemmas decomposing this goal into smaller, similar goals representing subexpressions of the final program. Section 3 provides the precise definition of these Hoare triples and gives more details on our syntax-directed search. We want to emphasize that this search can apply *any* lemma proving

such a Hoare triple, allowing the search to be extended with new implementation strategies.

This tree of deduction steps is produced automatically by a script in Ltac, Coq's Turing-complete tactic language, applicable to many other examples of the same ilk. New lemmas that prove Facade Hoare triples for new programs can be incorporated into this search to extend the compilation process with new strategies. Composed with the automation that comes before and after this stage, we have a fully automated, proof-generating pipeline from specifications to libraries of assembly code.

3. Example of compilation by synthesis

We begin by illustrating the compilation process on the example Fiat program from section 2:

```
rows ← shuffle($books ∩ {b | b.by = $author});
ret (map (λ row ⇒ row.Title) rows)
```

To compile this program, we synthesize a Facade program p according to the following specification[1]:

- p, when started in an initial state containing the arguments *$author* and *$books*, must be safe: i.e. p must not call functions without ensuring that their arguments are properly allocated and verifying the required preconditions; it may not access undefined variables; etc.
- p, when started in a proper initial state, must reach (if it terminates) a final state where all temporary data structures that p allocated have been deallocated, and where the variable `"out"` has one of the values allowed by the nondeterministic program p shown above.

The first condition ensures that the resulting program will be well-formed (and thus will meet the requirements of the Facade compiler); the second ensures that the program computes a correct value. We write $\boxed{args} \overset{p}{\underset{\varnothing}{\rightsquigarrow}} [\![\texttt{"out"} \mapsto \underline{p}]\!] :: \boxed{args}$ to summarize this specification; we read it as *the Facade program* p*, starting in any proper initial state, behaves correctly and stores a value permitted by the Fiat program* p *in the variable* `"out"`. One can think of it as a typical Hoare triple, with additional constraints on the pre- and postconditions to facilitate synthesis. Indeed, instead of the usual context where Hoare triples are used to help ensure that a program conforms to its specification, we use them here to drive the synthesis process.

The precise definition, shown in section 4, is constructed in such a way that Facade's semantics guarantee that any program respecting this specification will (if it terminates) correctly set the return value to one of the values permitted by the original Fiat program.

The final program that we wish to obtain is shown below. Note how the compiler chose to implement the nondeterministic selection of books shuffle(*$books* ∩ {b | b.by = *$author*}) using a call to the Facade function `BTree.find(books, author)`, and that it furthermore ensures that the resulting program does not leak any memory by deallocating the temporary `rows` variable.

```
rows ≜ BTree.find($books, $author);
out  ≜ StringList.new();
While (not TupleList.empty?(rows))
    row   ≜ TupleList.pop!(rows);
    title ≜ Tuple.get(row, 0);
    StringList.push!(out, title);
EndWhile
TupleList.delete!(rows);
StringList.reverse!(out);
```

Replacing p with our example Fiat program, we need to find a program p such that

$$[\![\texttt{"books"} \mapsto \texttt{ret } \$books]\!] :: [\![\texttt{"author"} \mapsto \texttt{ret } \$author]\!] \overset{p}{\underset{\varnothing}{\rightsquigarrow}} \left[\!\!\left[\texttt{"out"} \mapsto \begin{array}{l} rows \leftarrow \underline{\text{shuffle}}(\$books \cap \ldots); \\ \texttt{ret (map (}\lambda\ row \Rightarrow row\texttt{.Title) } rows\texttt{)} \end{array}\right]\!\!\right] :: [\![\texttt{"books"} \mapsto \texttt{ret } \$books]\!] :: [\![\texttt{"author"} \mapsto \texttt{ret } \$author]\!]$$

We use our first *compilation lemma* to connect the semantics of Fiat's bind operation (the ← operator of monads [22]) to the meaning of $\rightsquigarrow$, which yields the following synthesis goal:

$$[\![\texttt{"books"} \mapsto \texttt{ret } \$books]\!] :: [\![\texttt{"author"} \mapsto \texttt{ret } \$author]\!] \overset{p}{\underset{\varnothing}{\rightsquigarrow}} [\![\texttt{"tmp"} \mapsto \underline{\text{shuffle}}(\$books \cap \{\texttt{b | b.by = } \$author\}) \texttt{ as } r]\!] :: [\![\texttt{"out"} \mapsto \texttt{ret (map (}\lambda\ row \Rightarrow row\texttt{.Title) } r\texttt{)}]\!] :: [\![\texttt{"books"} \mapsto \texttt{ret } \$books]\!] :: [\![\texttt{"author"} \mapsto \texttt{ret } \$author]\!]$$

In this step, we have broken down the assignment (to a single variable) of a Fiat-level binding operation (`rows ← ...; ...`) into the construction of two variables: `"tmp"`, holding the intermediate list of selected authors, and `"out"`, holding the final result. The double-colon operator :: separates entries in a list of bindings of Facade variables to nondeterministic Fiat terms. Note that the description of Fiat states that we use is *causal*: we need to be able to track dependencies between different variables. Thus the ordering of the individual bindings matters: the Fiat term that we assign to `"out"` depends on the particular value chosen for `"tmp"` (which we bind locally as *r* with an `as` clause).

We then break down the search for p into the search for two smaller programs: the first (p_1) starts in the initial state (abbreviated to $\boxed{args}$) and is only concerned with the assignment to `"tmp"`; the second (p_2) starts in a state where `"tmp"` is already assigned and uses that value to construct the final result. Notice the similarity between the specification of p_1 and the original specification of p: just as p moved from an initial state to a state binding `"out"` to a Fiat expression, p_1 moves from an initial state to a state binding `"tmp"` to a Fiat expression.

$$\boxed{args} \overset{p_1}{\underset{\varnothing}{\rightsquigarrow}} [\![\texttt{"tmp"} \mapsto \underline{\text{shuffle}}(\$books \cap \ldots) \texttt{ as } r]\!] :: \boxed{args}$$

$$[\![\texttt{"tmp"} \mapsto \underline{\text{shuffle}}(\$books \cap \ldots) \texttt{ as } r]\!] :: \boxed{args} \overset{p_2}{\underset{\varnothing}{\rightsquigarrow}} [\![\texttt{"tmp"} \mapsto \underline{\text{shuffle}}(\$books \cap \ldots) \texttt{ as } r]\!] :: [\![\texttt{"out"} \mapsto \texttt{ret (map (}\lambda\ row \Rightarrow row\texttt{.Title) } r\texttt{)}]\!] :: \boxed{args}$$

At this point, a lemma about connecting the meaning of the nondeterministic selection of authors and the Facade-level `BTree.find` function tells us that `tmp` ≜

[1] In the following, underlined variables such as comp are Fiat computations, and slanted variables such as *r* are Gallina variables.

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

(a) 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 $\underline{\mathtt{v}}$.

$$\frac{\begin{array}{c}[\![\mathtt{ls} \mapsto \mathtt{ret}\ \ell]\!] :: t \overset{\mathtt{p_{init}}}{\underset{ext}{\rightsquigarrow}} [\![\mathtt{out} \mapsto \mathtt{ret}\ a_0]\!] :: [\![\mathtt{ls} \mapsto \mathtt{ret}\ \ell]\!] :: t \\ \forall\, h\ \underline{\mathtt{a}}\ \ell.\ [\![\mathtt{hd} \mapsto \mathtt{ret}\ h]\!] :: [\![\mathtt{out} \mapsto \underline{\mathtt{a}}]\!] :: t \overset{\mathtt{p_{body}}}{\underset{[\mathtt{ls} \mapsto \ell] :: ext}{\rightsquigarrow}} [\![\mathtt{out} \mapsto \underline{\mathtt{f}}\ \underline{\mathtt{a}}\ h]\!] :: t\end{array}}{[\![\mathtt{ls} \mapsto \mathtt{ret}\ \ell]\!] :: \mathtt{t} \overset{\mathtt{LOOP(p_{init},\ p_{body},\ ls)}}{\underset{ext}{\rightsquigarrow}} [\![\mathtt{out} \mapsto \mathtt{fold_L}\ \underline{\mathtt{f}}\ (\mathtt{ret}\ a_0)\ \ell]\!] :: t}\ \textsc{FoldL}$$

(b) The FoldL rule, connecting the functional reduction of ℓ with $\underline{\mathtt{f}}$ on initial value a_0 and the imperative computation of the same value using destructive iteration on a mutable list.

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

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

```
p_init ;
end ≜ List.empty?(ls);
While (not end)
  hd ≜ List.pop!(ls);
  p_body ;
  end ≜ List.empty?(ls);
EndWhile;
List.delete!(ls);
```

(d) The Facade `LOOP(p_init, p_body, ls)` macro.

Figure 2: Three rules used by our synthesizing compiler: Chomp, Bind, and FoldL. Bindings in Fiat states (*telescopes*) are written as $[\![\mathtt{key} \mapsto \mathtt{value}]\!]$, while regular key-value maps are written as `[key ↦ value]`. A fourth rule, If, is given in appendix A.

`BTree.find(&books, &author)` is a good choice for $\mathtt{p}_1$ (this is the *call rule* for `BTree.find`). We are therefore only left with $\mathtt{p}_2$ to synthesize: noticing the common prefix of the starting and ending states, we apply the *chomp* rule of Figure 2a, which allows us to set aside the common prefix and focus on the tail of the pre- and post-states, transforming the problem into

$$\forall\, r.\ r \in \underline{\mathtt{shuffle}}(\&books \cap \{\mathtt{b} \mid \mathtt{b.by} = \&author\}) \implies \boxed{args} \overset{\mathtt{p_2}}{\underset{[\text{"tmp"} \mapsto r]}{\rightsquigarrow}} [\![\text{"out"} \mapsto \mathtt{ret}\ (\mathtt{map}\ (\lambda\ \ldots)\ r)]\!] :: \boxed{args}$$

The additional mapping pictured under the $\rightsquigarrow$ arrow expresses an extra assumption that we can make about the starting and ending states: they must both map `"tmp"` to the same value r. The universal quantifier in the rule requires that the synthesized program be valid regardless of the particular value computed for `"tmp"`. It might seem that the process would allow us to construct a different program for each value of r, but it is fortunately not the case: $\mathtt{p}_2$ is an existential variable whose context does not contain r, so it cannot depend on it. In this form, after applying the universal introduction rule (Coq's `intro`), we can first rewrite `map` to $\mathtt{fold_L}$, at which point the synthesis goal matches the conclusion of the *fold*$_L$ rule: given a program $\mathtt{p_{init}}$ to initialize the accumulator and a program $\mathtt{p_{body}}$ to implement the body of the fold, the Facade program defined by the macro `LOOP(p_init, p_body, rows)` obeys the specification above. This gives us two new synthesis goals, which we can handle recursively, in a fashion similar to the one described above. Once these obligations have been resolved, we arrive at the desired Facade program.

4. Proof-generating extraction of nondeterministic functional programs

The extraction procedure outlined above is implemented in Ltac: starting from a possibly nondeterministic Fiat program, we apply sequences of rules tracking pre- and postconditions on the resulting program by using *Fiat states* describing classes of Facade program states. We start by describing the design of the Facade language and build on this description to explain the inner workings of the proof-producing extraction mechanism.

4.1 The Facade language

Facade[2] is an Algol-like untyped imperative language operating on Facade states, which are finite maps from variable names to Facade values. A Facade value is either a scalar (a stack-allocated integer) or a nonnull value of an ADT (heap-allocated, similar to an object in an OOP language). Syntactically, Facade includes standard programming constructs like assignments, conditionals, loops, and function calls. What distinguishes the language is its operational semantics, pictured partially in figure 3. First, that semantics follows that of Cito in supporting modularity by modeling calls to externally defined functions via preconditions and postconditions. That is, something of Hoare logic is built into the operational semantics, to serve as a bridge with code compiled from other languages. Second, *linearity* is baked into Facade's operational semantics, which enforce that every ADT value on the heap will be referred to by exactly one live variable (no aliasing and no leakage). This pattern of linearity simplifies reasoning about the formal connection to functional programs: if every object has at most one referent, then we can almost pretend that variables hold abstract values instead of pointers to mutable objects. In practice, we have not found this requirement overly constraining for our applications: one can automatically introduce copying when needed, or one can require the external ADTs to provide nondestructive iteration.

Facade successfully shields programmers from the complexities of low-level programming. The program semantics manipulates local-variable environments where ADTs are associated with high-level models. For instance, a finite set is modeled as a mathematical set, not as e.g. a hash table. A key

[2] A formal definition of the Facade language is presented in the technical report accompanying this paper.

$$\text{Statement } s ::= \texttt{Skip} \mid s\ \texttt{;}\ s \mid \texttt{x} \triangleq e \mid \texttt{If}\ e\ \texttt{Then}\ s\ \texttt{Else}\ s\ \texttt{EndIf} \mid \texttt{While}\ e\ s\ \texttt{EndWhile} \mid \texttt{x} \triangleq \texttt{Call l}(\overline{\texttt{x}})$$

$$\frac{[\![e]\!]_{\texttt{st}} = \texttt{Scalar(\_)} \qquad \texttt{st}(x) \neq \texttt{ADT(\_)}}{\Psi \vdash (\texttt{st}, \texttt{x} \triangleq \texttt{e}) \Downarrow [\texttt{x} \mapsto [\![e]\!]_{\texttt{st}}] :: \texttt{st}}\ \textsc{Assign}$$

$$\frac{\Psi(\texttt{l}) = \texttt{AX}(\mathsf{pre}, \mathsf{post}) \qquad \texttt{st}(x) \neq \texttt{ADT(\_)} \qquad \mathsf{pre}(\texttt{st}(\overline{\texttt{y}})) \qquad |\overline{v}| = |\overline{\texttt{y}}| \qquad \mathsf{post}(\texttt{st}(\overline{\texttt{y}}) \rhd \overline{v}, r)}{\Psi \vdash (\texttt{st}, \texttt{x} \triangleq \texttt{Call l}(\overline{\texttt{y}})) \Downarrow [\texttt{x} \mapsto r] :: [\overline{\texttt{y}} \mapsto \overline{v}] :: \texttt{st}}\ \textsc{CallAx}$$

Figure 3: Selected syntax and operational semantics of Facade.

parameter to the compiler soundness theorem is *a separation-logic abstraction relation, connecting the domain of high-level ADT models to mutable memories of bytes*. By picking different relations at the appropriate point in our pipeline, we can justify linking with different low-level implementations of high-level concepts. No part of our automated translation from Fiat to Facade need be aware of which relation is chosen, and the same result of that process can be reused for different later choices. This general approach to stateful encapsulation is largely inherited from Cito, though with Facade we have made it even easier to use.

Facade's operational semantics are defined by two predicates, $\Psi \vdash (\texttt{p}, \texttt{st})\downarrow$ and $\Psi \vdash (\texttt{p}, \texttt{st}) \Downarrow \texttt{st'}$, expressing respectively that the Facade program `p` will run safely when started in Facade state `st`, and that `p` may reach state `st'` when started from `st` (this latter predicate essentially acts as a big-step semantics of Facade). Both predicates are parameterized over a context Ψ mapping function addresses to their axiomatic specifications. The semantics is nondeterministic in the sense that there can be more than one possible `st'`.

Modularity is achieved by the fact that Facade's semantics contains the CallAx rule allowing a Facade program to call a function via its specification in Ψ. A function call's effect includes both the return value r and a list of output values, $\overline{v}$. These output values represent the result of in-place modification of input ADT arguments $\overline{\texttt{y}}$. When an input argument is an ADT value, the corresponding output value is an option: either another ADT value representing the new value of the input argument, or `None` to indicate that the input object has been deallocated. The ability of Facade function calls to modify or deallocate input arguments makes them manifestly impure, but we benefit from the flexibility to express memory effects such as combining, moving, and deallocation. A precondition is a predicate pre on the values assigned to the input arguments of the callee by the map `st`. A postcondition is a predicate post on these input values, output values $\overline{v}$, and return value r. The semantics prescribes that such a function call will nondeterministically pick a list of output values and a return value satisfying `post` and use them to update the relevant variables in the caller's postcall state.

Linearity is achieved by a set of syntactic and semantic provisions. For instance, variables currently holding ADT values cannot appear on the righthand sides of assignments, to avoid aliasing. They also cannot appear on the lefthand sides of assignments, to avoid losing their current payloads and causing memory leaks.

We have implemented a verified translation from Facade to Cito, and from there we reuse established infrastructure to connect into the Bedrock framework for verified assembly code[3]. Our translator is almost the identity function, as the syntaxes of the two languages are extremely close. Its soundness proof has more of the flavor of justifying a new type system for an existing language.

The starting point for our Facade-to-Cito compiler is a Facade module `m` containing the context Ψ of axiomatically specified functions imported by the module, a collection of labeled method implementations $\overline{\texttt{s}}$, and a corresponding collection of axiomatic pre- and post-conditions for those methods, $\{pre\}$ and $\{post\}$. As long as each method body is a valid implementation of its specification, the compiler is guaranteed to produce a Cito module whose methods also satisfy these specifications. This refinement is defined in a form morally equivalent to the $\rightsquigarrow$ relation introduced in the previous section. The resulting Cito module can be fed as input into the existing verified Cito compiler to produce a Bedrock assembly module with the same exported specification.

4.2 Fiat and Facade states

We connect Gallina's and Fiat's semantics to that of Facade by introducing a notion of *Fiat states*, helping us express constraints on the program being synthesized in a concise and systematic way and simplifying the synthesis by proof-search procedure. The intent is that, instead of augmenting a map relating names to Facade values with a number of propositions describing the relations and dependencies between these bindings, we may have a single data structure tracking the exact information required for compilation to succeed.

On the surface, Fiat states are similar to Facade states (they both describe collections of bindings). Facade states (frequently denoted as `st` in this paper) are unordered collections of bindings, with dependencies between bindings being expressed by additional propositions learned as one reasons about program transitions. We define Fiat states, on the other hand, as self-contained, dependently typed heterogeneous ordered lists of bindings (also called *telescopes* and denoted as st), where the head of each cons cell bundles a variable name and a propositionally described set of permissible values for that variable, while the tail of each cell is a function from values to telescopes (thus allowing later bindings to depend on earlier values). For example, the telescope $[\![$`"x"` $\mapsto$ any `as` $x]\!] :: [\![$`"y"` $\mapsto$ `ret (`x`, `x`)`$]\!]$ can be though of as describing all collections of names and values mapping `"x"` to a certain value x and `"y"` to a heap-allocated pair containing

[3] As a consequence, we also inherit some of the limitations of Bedrock's partial-correctness Hoare logic: for example, nonterminating programs are accepted as implementations of any specification, with one special case being early aborts when memory is exhausted. This issue is orthogonal to our techniques: we could easily add a return code to high-level memory-allocation methods if Bedrock gained the ability to report out-of-memory errors.

$$\frac{\forall\, \mathtt{k}\ v.\ \mathtt{st(k)} = \mathtt{Scalar}(v) \to \mathit{ext}\,(\mathtt{k}) = \mathtt{Scalar}(v) \qquad \forall\, \mathtt{k}\ v.\ \mathtt{st(k)} = \mathtt{ADT}(v) \leftrightarrow \mathit{ext}\,(\mathtt{k}) = \mathtt{ADT}(v)}{\mathtt{st} \lesssim \varnothing \uplus \mathit{ext}}\ \textsc{EqvStNil}$$

$$\frac{\mathtt{st(k)} = \mathtt{wrap}(v') \quad v' \in \underline{\mathtt{v}} \quad \mathtt{st\text{-}\{k\}} \lesssim (\mathit{st}\ v') \uplus \mathit{ext}}{\mathtt{st} \lesssim [\![\mathtt{k} \mapsto \underline{\mathtt{v}}\ \mathtt{as}\ v]\!] :: (\mathit{st}\ v) \uplus \mathit{ext}}\ \textsc{EqvStCons}$$

Figure 4: Equivalence relation on Fiat and Facade states.

two copies of x.

One can think of these telescopes as (deeply embedded) Fiat computations, with the significant difference that the monadic `Bind` constructor is extended to allow certain variables to be named. In addition, for convenience and to be able to implement the aforementioned *chomp* rule, Fiat states are extended with an unordered map (ext) from names to single values, which morally represents the Fiat variables for which a particular value has been picked among permissible values. A full Fiat state, thus, is composed of a telescope st and an extra collection of bindings ext, written $\mathit{st} \uplus \mathit{ext}$.

We relate Fiat states to Facade states using the ternary predicate $\mathtt{st} \lesssim \mathit{st} \uplus \mathit{ext}$ defined in figure 4. This relation ensures that the values assigned to each variable in the Facade state `st` are compatible with the bindings described in the Fiat state $\mathit{st} \uplus \mathit{ext}$. Each named variable in Fiat states is annotated with a function `wrap` describing how to inject values of its type in and out of the representation type used at the Facade level. The predicate also ensures that the values chosen for each of the Facade variables are compatible with choices made for previous bindings.

Because Facade's manual memory-management model does not allow us to leak ADTs, we require that all bindings pointing to ADTs in `st` be reflected in $\mathit{st} \uplus \mathit{ext}$, and vice versa. For scalars, on the other hand, we only require that bindings in $\mathit{st} \uplus \mathit{ext}$ be mirrored in `st`. Intuitively, this means that we are allowed to forget about previously allocated scalar values, but we must track ADTs precisely until we deallocate them. Examples of this relation are given in appendix B.

4.3 Proof-generating extraction by synthesis

Armed with this predicate, we are ready to give the full definition of $\mathit{st} \overset{\mathtt{p}}{\underset{\mathit{ext}}{\rightsquigarrow}} \mathit{st'}$, shown below:

$$\begin{cases} \forall\, \mathtt{st}.\ \mathtt{st} \lesssim \mathit{st} \uplus \mathit{ext} \implies (\mathtt{p}, \mathtt{st}) \downarrow \\ \forall\, \mathtt{st}\ \mathtt{st'}.\ \mathtt{st} \lesssim \mathit{st} \uplus \mathit{ext} \wedge (\mathtt{p}, \mathtt{st}) \Downarrow \mathtt{st'} \\ \qquad\qquad \implies \mathtt{st'} \lesssim \mathit{st'} \uplus \mathit{ext} \end{cases}$$

This specification reads as follows[4]:

- For all starting Facade states `st`, if `st` is in relation with the Fiat state st extended by ext, then it is safe to run the Facade program `p` from state `st`.
- For all starting and ending Facade states `st` and `st'`, if `st` is in relation with the Fiat state st extended by ext, and if running the Facade program `p` starting from `st` may produce the Facade state `st'`, then `st'` is in relation with the Fiat state $\mathit{st'}$ extended by ext.

This definition is enough to concisely and precisely phrase the three types of lemmas required to synthesize Facade programs:

- Properties of the $\rightsquigarrow$ relation (most importantly the CHOMP rule of Figure 2a), used to drive the proof search and provide the extraction architecture
- Connections between the $\rightsquigarrow$ relation and Fiat's semantics (for example the BIND rule of Figure 2c), used to reduce extraction of Fiat programs to that of Gallina programs
- Connections between Fiat and Facade, through the $\rightsquigarrow$ relation (such as the FOLDL rule of Figure 2b). This last category is the one in which user extensions are found: additional lemmas extend the scope of the compiler and broaden the range of source programs that the synthesizing compiler is able to handle.

With these lemmas, we can phrase certified extraction as a proof-search problem that can be automated effectively. Starting from a Fiat computation $\underline{\mathtt{f}}\ x_1 \ldots x_n$ mixing Gallina code with calls to external ADTs, we generate a specification $\uparrow\!\underline{\mathtt{f}}\!\uparrow$ based on the $\rightsquigarrow$ predicate (which itself is defined in terms of Facade's operational semantics):

$$\uparrow\!\underline{\mathtt{f}}\!\uparrow \triangleq \exists\, \mathtt{p}.\ \forall\, x_1 \ldots x_n. \qquad (1)$$
$$[\![\text{"}\mathtt{x}_1\text{"} \mapsto \mathtt{ret}\ x_1]\!] :: \ldots \overset{\mathtt{p}}{\underset{\varnothing}{\rightsquigarrow}} [\![\text{"}\mathtt{out}\text{"} \mapsto \underline{\mathtt{f}}\ x_1 \ldots x_n]\!]$$
$$\ldots :: [\![\text{"}\mathtt{x}_n\text{"} \mapsto \mathtt{ret}\ x_n]\!]$$

From this starting point, extraction proceeds by analyzing the joint shapes of the pre and post-states to determine applicable compilation rules, which are then applied to break down and simplify the post-state. This stage explains why we chose strongly constrained representations for pre and post-states: unlike a verification task, where the program's source drives the computation of verification conditions, we do not have source code to guide the process; instead, we use the shape of the pre- and postconditions to drive the derivation of the program (proper care in designing the compilation rules and their preconditions obviates the need for backtracking).

In practice, this pattern matching is implemented by a collection of matching functions written in Ltac which, given pre- and postconditions, either return without changing the goal, solve the current goal by applying a synthesis lemma, or produce a new goal by applying a compilation lemma of the form shown in figure 2. For example, one of these matching rules looks at the first binding in the precondition's telescope and searches for use of the corresponding value in the postcondition. If the value is not used, then the program being synthesized is replaced by a sequence of two operations: the deallocation of the head variable in the precondition, followed by an unknown program to be synthesized with a precondition not mentioning the recently deallocated variable.

Our extraction architecture is extensible: the main loop exposes hooks that users can rebind to call their own matching rules, providing support for more Gallina or Fiat forms. Examples of such rules are provided in section 6.1. Of course,

[4] The context of available functions Ψ remains implicit in these definitions.

since our focus is not on writing a general Gallina extraction mechanism (but instead on extracting Gallina EDSLs supporting nondeterministic programs), these rules do not cover all possible Gallina programs: instead, when the compiler does not know how to implement a given construct $\underline{\mathsf{C}}$, compilation stops and the user is presented with a new synthesis goal of the general form $pre \overset{?}{\underset{ext}{\rightsquigarrow}} [\![\mathtt{k} \mapsto \underline{\mathsf{C}}]\!] :: post$ (depending on the particular goal, this program may or may not admit an efficient imperative implementation: though there are no fundamental restrictions on which programs may be compiled using our technique, some of them might e.g. require garbage collection). In our experience, debugging proof search and adding support for new constructs is relatively easy, though it does require strong familiarity with Coq (appendix C).

5. The complete proof-generating pipeline

The components presented in the previous section form the final links in an automated pipeline lowering high-level specifications to certified Bedrock modules, whose correctness is guaranteed by Theorem 1.

Starting from a Fiat ADT specification $\underline{\mathsf{ADT}}_{\mathtt{spec}}$ (a collection of high-level method specifications $\underline{\mathsf{m}}_{\mathtt{spec}}$, as shown in figure 6a), we obtain by refinement under a relation $\approx$ a Fiat ADT implementation $\underline{\mathsf{ADT}}_{\mathtt{impl}}$ (a collection of nondeterministic functional programs $\underline{\mathsf{m}}_{\mathtt{impl}}$, as shown in figure 6b). Each method of this implementation is assigned an operational specification $\lceil \underline{\mathsf{m}}_{\mathtt{impl}} \rceil$ (equation 1), from which we extract (using proof-producing synthesis) a verified Facade implementation $\mathtt{m}_{\mathtt{impl}}$ (section 4.3) that calls into a number of external functions (Ψ, figure 3b), as shown in figure 6c.

Finally, we package the resulting Facade methods into a Facade *module*. This module imports Ψ (i.e. it must be linked against implementations of the functions in Ψ) and exports axiomatic specifications straightforwardly lifted *from the original high-level specifications* into Facade-style axiomatic specifications (of the style demonstrated in the call rule of figure 3b): for each high-level specification $\underline{\mathsf{meth}}_{\mathtt{spec}}$, we export the following (written $\lceil \underline{\mathsf{meth}}_{\mathtt{spec}} \rceil$):

```
Pre ≜ λ args ⇒ ∃ r_S r_I xs.
  r_S ≈ r_I ∧ args = [r_I] ++ xs
Post ≜ λ args ⇒ ∃ r_S r_I r'_S r'_I v xs.
  r_S ≈ r_I ∧ (r'_S, v) ∈ meth_spec r_S xs ∧
  r'_S ≈ r'_I ∧ args = [(r_I, r'_I)] ++ (zip xs xs)
```

A *generic* proof guarantees that the operational specifications $\lceil \underline{\mathsf{meth}}_{\mathtt{impl}} \rceil$ used to synthesize Facade code are indeed a refinement of axiomatic specifications $\lceil \underline{\mathsf{meth}}_{\mathtt{spec}} \rceil$ exported by our Facade module. Compiling this Facade module via our new formally verified Facade compiler produces a correct Bedrock module, completing Theorem 1 (a figure summarizing this theorem is supplied as appendix D):

THEOREM 1. *From a valid refinement* $\underline{\mathsf{ADT}}_{\mathtt{impl}}$ *of a Fiat ADT specification* $\underline{\mathsf{ADT}}_{\mathtt{spec}}$ *with methods* $\underline{\mathsf{meth}}_{\mathtt{impl}}$ *and* $\underline{\mathsf{meth}}_{\mathtt{spec}}$ *and a set of Facade programs synthesized from each* $\lceil \underline{\mathsf{meth}}_{\mathtt{impl}} \rceil$*, we can build a certified Bedrock module whose methods satisfy the axiomatic specifications* $\lceil \underline{\mathsf{meth}}_{\mathtt{spec}} \rceil$.

Note how, although the Facade code is derived from the *refined* Fiat implementations, the final Bedrock module satisfies the *original, high-level* Fiat specifications. The final compiled module clearly specifies its external dependencies Ψ, and Bedrock assembly code along with proofs of conformance to the axiomatic specifications of these methods must be provided for each of them as part of the final *linking* phase. Once the linking phase (which happens entirely inside of Coq) is complete, we obtain a closed, executable Bedrock module, exposing an axiomatic specification directly derived from the original, high-level ADT specification. Our implementation links against hand-verified implementations of low-level indexing structures (nested binary search trees of tuples implemented and verified with the Bedrock Structured Programming System [1], an extensible C-like language with integrated specification annotations), though it would be possible to use any compiler emitting Bedrock assembly code.

Trusted base Our derivation is assumption-free in the Calculus of Inductive Constructions extended with Ensemble Extensionality (a restriction of functional extensionality to functions describing sets) and Axiom K [19]. Our trusted base (beyond an assembler) 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), and hand-written assembly wrappers for the extracted methods (∼50 lines of assembly code). Details about our implementation and proof styles are given in appendix E.

6. Evaluation

We evaluated our extraction strategy using three benchmarks: a collection of small programs stressing the proof-producing extraction part of our pipeline; a family of binary encoders marshalling packets for network transmission; and a process-scheduler ADT from Fiat's query-structure domain exercising our entire pipeline, from specifications to assembly.

6.1 Microbenchmarks

We started out by extracting a collection of small Gallina programs manipulating machine words, all handled by our default library of compilation rules; extraction takes a few seconds for each program[5].

We then extended the compiler by providing three new pieces: a simple specification for an external function $\underline{\mathsf{any}}$ returning an arbitrary number, a Bedrock-style specification for an external `std.rand` Facade function, and a verified *call rule* connecting these two specs using the $\rightsquigarrow$ relation. By hooking this rule into our compiler, we were then able to extract a variety of examples with simple nondeterministic

[5] All examples, and the corresponding outputs, are included in the '`MicrobenchmarksAnnotated.v`' file attached to this submission; the benchmarks can be also run interactively by stepping through `src/CertifiedExtraction/Benchmarks/MicrobenchmarksAnnotated.v`

```
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 5: 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).

choices of numbers. In total, extending the compiler to support this external function (whose behavior could not be obtained in pure Gallina) required about 20 lines of specifications, 5 lines of proofs, and 10 lines of compilation script.

Another 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 (figure 5).

These examples illustrate that the source language of our extraction engine is a fluid, extensible DSL, including a subset of Gallina and encompassing many monadic programs beyond pure Gallina (nondeterministic Fiat programs). As a more advanced example we extended the compiler to translate maps and folds into destructive while loops (figure 2b), which allowed us to compile examples like filtering lists or flattening lists of lists (using nested while loops).

In total, our microbenchmarks suite contains about 25 programs manipulating machine words, lists, and nested lists thereof, with optional nondeterministic picks (any). Examples range from simple operations such as performing basic arithmetic, allocating data structures, calling compiler intrinsics, or sampling arbitrary numbers to more complex operations involving sequence manipulations, like reversing, filtering, reducing (e.g. reading in a number written as a list of digits in a given base), flattening, and duplicating or replacing elements. A detailed example of such a microbenchmark is given in appendix F1.

6.2 Binary Encoders

Our second set of benchmarks provides a more realistic demonstration of the capabilities of our extraction engine by compiling a family of binary encoders written in a suitable EDSL. These programs serialize in-memory data into a binary format suitable for network transmission. Details are given in appendix F2.

6.3 Relational Queries

As a final, comprehensive benchmark of our pipeline in real-life conditions, we targeted the query-structure ADT library of the Fiat paper [2] as well as an ADT modeling process scheduling inspired by Hawkins et al.. This benchmark starts from high-level Fiat specifications (as shown in figure Figure 6a) and outputs a closed Bedrock module, linked against a hand-verified nested-binary-tree implementation. The intermediate phases of this process are summarized in figure 6.

Starting from Fiat specifications, we use the existing Fiat framework to derive a collection of nondeterministic Fiat programs (one per ADT method, as demonstrated in Figure 6b). These computations, mixing pure Gallina and nondeterministic computations (modeling uses of a bag data structure), are then fed into our proof-producing extractor. Extraction to Facade is fully automatic; it draws from the default pool of extraction lemmas (about conditionals, constants, arithmetic operations, etc.) and from a few bag-specific lemmas that we hooked into the compiler (these manually verified *call rules* connect the pure bag specifications used in Fiat sources to Bedrock-style specifications of mutable binary search trees using the $\rightsquigarrow$ relation; appendix G gives a detailed example). The result is a Facade module that exports axiomatically specified methods proven to conform to the high-level specifications of the original Fiat ADT (figure Figure 6c). Each dependency declared by the Facade module is expressed as an axiomatic specification and resolved upon linking with a suitable implementation.

In line with our ideal of facilitating the use of domain-specific optimizations providing better performance, initial extraction results led us to extend Fiat refinements to use `revmap` instead of `map` where allowed by the specification (queries can return results in any order), as `revmap` (unlike `map`) can be implemented in a single pass over its input list in constant stack space.

After extracting the ADT (this 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), we compiled the resulting programs using the new Facade-to-Bedrock compiler: this produced an assembly library with proofs of partial correctness against the original SQL-flavored specifications. Finally, we produced a closed, executable binary by soundly linking this program against a hand-crafted, verified implementation of nested binary trees (written in Bedrock assembly

```
Definition SchedulerSpec: ADT ≜
 QueryADTRep SchedulerSchema {
  Def Init: rep ≜ empty,

  Insert a new process, failing if newpid already exists.
  Def Spawn (r: rep) (newpid cpu st: W32 ): 𝔹 ≜
   Insert (<pid:: newpid, state:: st, cpu:: cpu>)
     into r.Processes,

  Find all processes in a state st and return their PIDs.
  Def Enumerate (r: rep) (st: W32 ): list W32  ≜
   procs ← For (p in r.Processes)
           Where (p.state = st)
           Return (p.pid);
   ret (r, procs),

  Find a process by PID and return its CPU time.
  Def GetCPUTime (r: rep) (id: W32 ): list W32  ≜
   proc ← For (p in r.Processes)
          Where (p.pid = id)
          Return (p.cpu);
   ret (r, proc) }.
```

(a) The original Fiat specification of a process scheduler. The refinement process derives an efficient functional implementation of this specification by implementing it using nested trees keyed on the process ID, followed by the process state.

```
Spawn
a ← bfind r (_, $newpid, _);
if length (snd a) = 0 then
  u ← binsert (fst a) [$newpid, $state, $cpu];
  ret (fst u, true)
else
  ret (fst a, false)

Enumerate
a ← bfind r ($state, _, _);
ret (fst a, revmap (λ x ⇒ GetAttr x 0) (snd a))

GetCPUTime
a ← bfind r (_, $id, _);
ret (fst a, revmap (λ x ⇒ GetAttr x 2) (snd a))
```

(b) The output of Fiat, after refining the specifications presented in Figure 6a. Notice the use of bfind and binsert, two nondeterministic methods of a bag ADT. In this example, the bag data structure that bfind depends on is expected to provide fast lookups by state and then by ID.

```
Enumerate
procs ≜ BTree.findFirst(rep, $state);
return ≜ List[W32 ].new();
test ≜ List[Tuple].empty?(procs);
While (not test)
    head ≜ List[Tuple].pop!(procs);
    pos ≜ 0;
    head' ≜ Tuple.get(head, pos);
    size ≜ 3;
    Call Tuple.delete!(head, size);
    Call List[W32 ].push(ret, head');
    test ≜ List[Tuple].empty?(procs);
EndWhile;
Call List[Tuple].delete!(procs);
```

(c) Output of the compilation from Fiat to Facade for the Enumerate method. The low-level data structure that we will link against is a nested tree, indexed by state and then by process ID. The call to findFirst returns a list of all processes in a particular state. GetCPUTime has a similar shape but is implemented using a skip-scan (or "loose index scan") on the first level of the nested tree (process states), followed by a search on the second level of the tree (process IDs).

Figure 6: Different stages of the process-scheduler compilation example (see also the annotated, anonymized 'ProcessScheduler.v' file attached to this submission).

for performance). We ran the verified programs to confirm experimentally that they behaved as expected, both in terms of raw performance and of scaling.

Figure 7 presents the results of our experimental validation. We compare our own verified implementation ("Fiat") against the corresponding SQL queries executed by SQLite 3.8.2 (using an in-memory database) and PostgreSQL 9.3.11 ("PG"). In all cases, the data is indexed by (state, PID) to allow for constant-time Enumerate queries (the number of active processes is kept constant) and logarithmic-time GetCPUTime queries (assuming a B-tree–style index and skip-scans).

Our implementation behaves as expected: it beats SQLite and PostgreSQL by 1.5 and 2.5 orders of magnitude respectively on GetCPUTime, and competes honorably with SQLite (while beating PostgreSQL by one order of magnitude) on Enumerate. Notice the red curves on the graph: we had to tweak the queries fed to PostgreSQL and SQLite to obtain good GetCPUTime performance: without an explicit "`$state IN (0, 1)`" clause, they both missed the skip-scan opportunity and exhibited rapidly degrading, asymptotically suboptimal linear-time behavior shown in red.

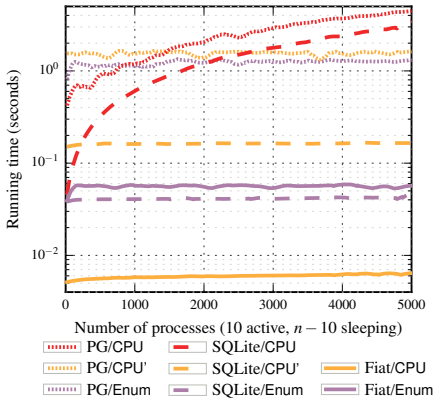

Figure 7: Scalability and performance benchmarks for the process-scheduler example. For increasingly large collections of processes, we run 20,000 Enumerate queries to locate the 10 active processes, followed by 10,000 GetCPUTime queries for arbitrary process IDs.

7. Related work

Closely related to our work is a project by Lammich [7], which uses Isabelle/HOL to refine functional programs to an embedded imperative language that depends on garbage collection (unlike Facade). Lammich's tool has been applied to various relatively complex algorithms, whereas our focus is on fully automatic derivation from highly regular specs, along the lines of what real-world programmers write regularly. Both approaches use some form of linearity checking to bridge the functional-imperative gap: Lammich uses separation logic [15] and axiomatic semantics, while we apply Facade's lihter-weight approach (decidable syntactic checks applied after the fact, with no explicit pointer reasoning). Crucially, while Lammich's tool only targets Imperative/HOL,

our pipeline allows integration of low-level programs with verified libraries compiled from any language of the Bedrock ecosystem. Furthermore, we have successfully integrated our translation into an automated proof-generating pipeline from relational specifications to executable assembly code — as far as we know, no such pipeline has been presented before.

Another closely related project by Myreen and Owens [12] focuses on extracting terms written in the purely functional subset of HOL4's logic into CakeML programs (CakeML is a dialect of ML). The main differences with our pipeline, beyond language choices, are in optimization opportunities, extensibility, and external linking abilities. Indeed, while Myreen and Owens's project bridges a relatively narrow gap (between two functional languages with expressive type systems and automatic memory management), our extraction procedure connects two very different languages, opening up many more opportunities for optimizations (including some related to memory management). We expose these opportunities to our users by letting them freely extend the compiler based on their domain-specific optimization knowledge. Finally, our approach allows us to compile partially refined Fiat programs whose nondeterministic fragments describe dependencies on externally implemented methods.

One last related project is the compiler of the COGENT language [14]. COGENT's source language is very close to Facade (it allows for foreign calls to axiomatically specified functions, but it does not permit iteration or recursion — except through foreign function calls), and its compiler also produces low-level code without a garbage collector. Our projects, however, differ both in architecture and in spirit: COGENT is closer to a traditional verified compiler, producing consecutive embeddings of a source program (from C to a shallow embedding in Isabelle/HOL) and generating equivalence proofs connecting each of them. A key contribution of COGENT is its linear type system, which establishes memory safety when connecting deep and shallow embeddings. Our approach favors extensibility over completeness, using the entirety of Gallina as our source language and collections of lemmas to justify the compilation of each source construct.

We draw further inspiration from a number of other efforts:

Program extraction Program extraction (a facility offered by Coq and other proof assistants) is a popular way of producing executable binaries from verified code. Extractors are rather complex programs, subjected to varying degrees of scrutiny: for example, the theory behind Coq's extraction was mechanically formalized and verified [10], but the corresponding concrete implementation itself is unverified. In contrast, our pipeline ensures that initial specifications are fully preserved down to the generated Bedrock code.

Compiler verification Our compilation strategy allows Fiat programs to depend on separately compiled libraries. This contrasts with verified compilers like CakeML [6] or CompCert [9]: in the latter, correctness guarantees only extend to linking with modules written in CompCert C and compiled with the same version of the compiler. Recent work [18] generalized these guarantees to cover cross-language compilation, but these developments have not yet been used to perform functional verification of low-level programs assembled from separately verified components.

An alternative approach, recently used to verify an operating-system kernel [16], is to validate individual compiler outputs. This is particularly attractive as an extension of existing compilers, but it generally falls short when trying to verify complex optimizations, such as our high-level selection of algorithms and data structures. In the same vein, verified compilers often rely on unverified programs to solve complex problems such as register allocation, and use a verified checker to validate solutions. In our context, the solver is the proof-producing extraction logic, and the verifier is Coq's kernel: our pipeline produces proofs that witness the correctness of the resulting Facade code.

Extensible compilation Multiple research projects let users add optimizations to existing compilers. Some, like Racket [21], do not focus on verification. Others, like Rhodium [8], let users phrase and verify transformations using DSLs. Unfortunately, most of these tools are unverified and do not provide end-to-end guarantees. One recent exception is XCert [20], which lets CompCert users soundly describe program transformations using an EDSL. Our approach is similar insofar as we assemble domain-specific compilers from collections of verified rewritings.

Program synthesis Our approach of program generation via proofs follows in the deductive-synthesis tradition started in the 1980s [11]. We use the syntactic structure of our specialized pre- and postconditions to drive synthesis: this idea of strongly constraining the search space is inherited from the syntax-guided approach pioneered in the *Sketch* language [17]. That family of work uses SMT solvers where we use a proof assistant, offering more baseline automation with less fundamental flexibility.

Formal decompilation Instead of deriving low-level code from high-level specifications, some authors have used HOL-family proof assistants to translate unverified low-level programs (in assembly [13] or C [4]) into high-level code suitable for verification. Decompilation is an attractive approach for existing low-level code, or when compiler verification is impractical.

8. Conclusion

The synthesis-based extraction techniques presented in this paper are a convenient and lightweight approach for generating certified extracted programs, reducing the trusted base of verified programs to little beyond the proof assistant's kernel. We have shown our approach to be suitable for the extraction of DSLs embedded in proof assistants, using it to compile a series of microbenchmarks and to do end-to-end proof-generating derivation of assembly code from SQL-style specifications. Crucially, the latter derivations work via link-

ing with verified implementations of assembly code that our derivation pipeline could never produce directly. To ease this transition, we developed Facade, a new language designed to facilitate reasoning about memory allocation in synthesized extracted programs. In the process, we have closed the last gap in the first automatic and mechanically certified translation pipeline from declarative specifications to assembly-language libraries, supporting user-guided optimizations and parameterization over abstract data types that may be implemented, compiled, and verified using arbitrary languages and tools.

References

[1] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proc. ICFP*, pages 391–402. ACM, 2013.

[2] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proc. POPL*, 2015.

[3] E. W. Dijkstra. A constructive approach to the problem of program correctness. Circulated privately, Aug. 1967. URL http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD209.PDF.

[4] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In Lennart Beringer and Amy Felty, editor, *International Conference on Interactive Theorem Proving*, pages 99–115, Princeton, New Jersey, USA, aug 2012. Springer Berlin / Heidelberg. doi: 10.1007/978-3-642-32347-8_8.

[5] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011.

[6] R. Kumar, M. O. Myreen, S. Owens, and M. Norrish. CakeML: A verified implementation of ML. In *Proc. POPL*. ACM, 2014.

[7] P. Lammich. Refinement to Imperative/HOL. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving*, volume 9236 of *Lecture Notes in Computer Science*, pages 253–269. Springer International Publishing, 2015. ISBN 978-3-319-22101-4. doi: 10.1007/978-3-319-22102-1_17.

[8] S. Lerner, E. Rice, T. Millstein, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. Technical report, 2004.

[9] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54. ACM, 2006.

[10] P. Letouzey. A new extraction for Coq. In *Proc. TYPES*. Springer-Verlag, 2003.

[11] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, Jan. 1980. ISSN 0164-0925. doi: 10.1145/357084.357090. URL http://doi.acm.org/10.1145/357084.357090.

[12] M. O. Myreen and S. Owens. Proof-producing synthesis of ML from higher-order logic. In *International Conference on Functional Programming (ICFP)*. ACM, 2012.

[13] M. O. Myreen, M. J. C. Gordon, and K. Slind. Decompilation into logic – Improved. In G. Cabodi and S. Singh, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 78–81. IEEE, 2012.

[14] L. O'Connor, C. Rizkallah, Z. Chen, S. Amani, J. Lim, Y. Nagashima, T. Sewell, A. Hixon, G. Keller, T. C. Murray, and G. Klein. COGENT: certified compilation for a functional systems language. *CoRR*, abs/1601.05520, 2016. URL http://arxiv.org/abs/1601.05520.

[15] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS*, pages 55–74. IEEE Computer Society, 2002.

[16] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proc. PLDI*, pages 471–482. ACM, 2013.

[17] A. Solar-Lezama. The sketching approach to program synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 4–13, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-10671-2. doi: 10.1007/978-3-642-10672-9_3.

[18] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *Proc. POPL*. ACM, 2015.

[19] T. Streicher. *Investigations into intensional type theory*. Habilitation thesis, 1993.

[20] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *Proc. PLDI*, pages 111–121. ACM, 2010.

[21] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proc. PLDI*, pages 132–141. ACM, 2011.

[22] P. Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[23] P. Wang, S. Cuellar, and A. Chlipala. Compiler verification meets cross-language linking via data abstraction. In *Proc. OOPSLA*, pages 675–690. ACM, 2014.