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

Clément Pit-Claudel[1][0000−0002−1900−3901], Peng Wang[2], Benjamin Delaware[3], Jason Gross[1][0000−0002−9427−4891], and Adam Chlipala[1]

[1] MIT CSAIL, Cambridge, MA 02139, USA
`{cpitcla,jgross,adamc}@csail.mit.edu`
[2] Google, Mountain View, CA 94043, USA
`wangpeng@google.com`
[3] Purdue University, West Lafayette, IN 47907, USA
`bendy@purdue.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 with overall full-functional-correctness proofs in separation logic.

1 Introduction

The general area of correct-by-construction code generation is venerable, going back at least to Dijkstra's work in the 1960s [5]. Oftentimes, solutions offer a strict subset of the desiderata of generality, automation, and performance of synthesized code. This paper presents the final piece of 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

N. Peltier and V. Sofronie-Stokkermans (Eds.): IJCAR 2020, LNAI 12167, pp. 1–19, 2020. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-51054-1_7

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. Our pipeline is *foundational*: it produces a fully linked assembly program represented as a Coq term with a proof that it meets the original high-level specification.

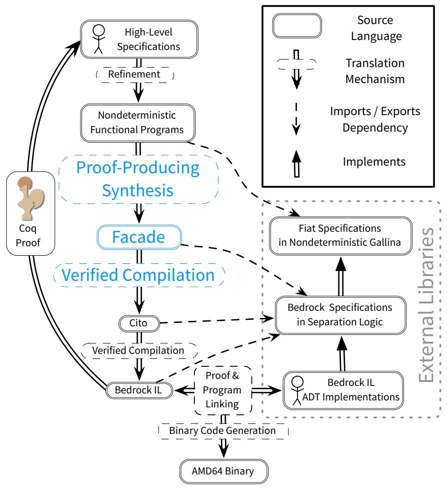
Source Language
Translation Mechanism
Imports / Exports Dependency
Implements
High-Level Specifications
Refinement
Nondeterministic Functional Programs
Proof-Producing Synthesis
Facade
Verified Compilation
Cito
Verified Compilation
Bedrock IL
Proof & Program Linking
Binary Code Generation
AMD64 Binary
Coq Proof
Fiat Specifications in Nondeterministic Gallina
Bedrock Specifications in Separation Logic
Bedrock IL ADT Implementations
External Libraries

Fig. 1: The full pipeline, with this work's contributions in blue. Stick figures indicate user-supplied components.

Our complete toolchain uses Fiat [4] to refine high-level specifications of abstract data types (ADTs) into nondeterministic functional programs depending on external data structures (expressed in a shallowly embedded Gallina DSL), then soundly extracts these programs to an imperative intermediate language (Facade) using a novel proof-generating extraction procedure. The resulting programs are then translated into the Cito [29] language by a newly written compiler, backed by a nontrivial soundness argument bridging two styles of operational semantics. A traditional verified compiler produces efficient Bedrock assembly [3] from the Cito level, which we soundly link 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 (extraction) to present the first mechanically certified automatic translation pipeline from declarative specifications to efficient assembly programs, as shown in Fig. 1.

In the original Fiat system, specifications were highly nondeterministic programs, and final implementations were fully deterministic programs obtained by repeatedly *refining* the specification, eventually committing to a single possible result. As a consequence, the generated code committed to a particular deterministic (and pure) implementation of external ADTs and functions that it relied on, reducing flexibility, optimization opportunities, and overall performance. Additionally, the final step in previous work using Fiat was to *extract* this code directly to OCaml, using Coq's popular but unverified extraction mechanism. Unfortunately, this meant that correctness of the compiled executable depended 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 decreased the confidence that users can place in programs synthesized by Fiat, and more generally in all programs extracted from Gallina code.

Our work overcomes these issues via a novel approach to extraction that is both *extensible* and *correct* and produces *efficient, stateful low-level code* from nondeterministic functional sources. The process runs within Coq, produces assembly code instead of OCaml code, and supports linking with handwritten or separately compiled verified assembly code.

Instead of refining specifications down to a fully deterministic Gallina program, as the original Fiat system did, we allow Fiat's final output to incorporate nondeterminism. These choices are resolved at a later stage by interpreting the nondeterminism as a postcondition specification in Hoare logic and linking against assembly code proven to meet that specification. Nondeterminism at runtime, which is not normally present in Gallina programs, is essential to support code derivation with flexible use of efficient low-level data structures. For example, if we represent a database with a type of binary trees that does not enjoy canonical representations, the same data may admit multiple concrete representations, each corresponding to a different ordering of results for an operation enumerating all database records.

Unlike certified compilers like CompCert [13] or CakeML [9], we do not implement our translator in the proof assistant's logic and prove it sound once and for all. Instead, we use proof-generating extraction: we phrase the translation problem in a novel sequent-calculus-style formulation that allows us to apply all of Coq's usual scriptable proof automation. The primary reason is that we want to make our compiler *extensible* by not committing to a specific compilation strategy: in our system, programmers can teach the compiler about new verified low-level data structures and code-generation strategies by introducing new lemmas explaining how to map a Gallina term to a particular imperative program[4]. Our automation then builds a (deeply embedded) syntax tree by repeatedly applying lemmas until the nondeterministic functional program is fully compiled. The many advantages of this approach (extensibility, ease of development, flexibility, performance, 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.

To summarize the benefits of our approach:

– It is lightweight: it does not require reifying the entirety of Gallina into a deeply embedded language before compiling. Instead, we use Coq's tactic 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 assemble a customized compiler by supplying their own compilation lemmas to extend the source language or improve the generated code.

[4] In fact, nondeterministic choices *cannot* be compiled systematically, as they can represent arbitrary complexity. Additionally, a proof-producing approach lets us elegantly bypass the issue of self-reference, since our original programs are shallowly embedded.

– It is well-suited to compiling EDSLs: we support nondeterminism in input programs (standard extraction requires deterministic code).
– It allows us to link against axiomatically specified foreign functions and data structures, implemented and verified separately.
– It compiles to a relatively bare language with explicit memory management.

To demonstrate the applicability of this approach, section 6 presents a set of microbenchmarks of Fiat programs manipulating variables, conditions, and nested lists of machine words, as well as a more realistic example of 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 available online at https://github.com/mit-plv/fiat/tree/ijcar2020.

2 A brief outline of our approach

We begin with an example of the pipeline in action. Below are an SQL-style query finding all titles by an author and a Fiat-generated implementation (right):

```
SELECT title FROM Books     | rows ← IndexedByAuthor.bfind $books $author;
WHERE Books.by = $author    | ret (map (λ row ⇒ row.Title) rows)
```

The generated code relies on a Fiat module `IndexedByAuthor`, which is not an executable implementation of the required functionality; rather, it specifies certain methods nondeterministically, implying that bfind returns the expected rows in some *undetermined* order. The order may even be different for every call, as might arise, for instance, with data structures like splay trees that adjust their layout 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 [3], obtained by extracting to a new language, *Facade*, built as a layer on top of the Cito C-like language [29], and then compiling to Bedrock.

The output for our running example might look like the code on the right. Note that this code works directly with pointers to heap-allocated mutable objects, handling all memory management by itself, including for intermediate values. The general `IndexedByAuthor` interface has been replaced with calls to a concrete module `BTree` providing binary search trees of tuples, and the call to `map` became an imperative loop. We implement and verify `BTree` in Bedrock assembly, and then we link code and proofs to obtain a binary and an end-to-end theorem guaranteeing full functional correctness of assembly libraries, for code generated automatically from high-level specifications.

```
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);
```

The heart of our contribution is spanning the gap from *nondeterministic functional programs* (written in Gallina) to *imperative low-level programs* (written in Facade) using an extensible, proof-generating framework. We phrase this derivation problem as one of finding a proof of a 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 $\underline{\text{IndexedByAuthor.bfind}}$ (subsection 4.3 defines these triples precisely).

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

The actual implementation of ?1 is found by applying lemmas to decompose this goal into smaller, similar goals representing subexpressions of the final program. These lemmas form a tree of deduction steps, produced automatically by a syntax-directed compilation script written in Coq's *Ltac* tactic language. Crucially, the derivation implemented by this script can include *any* adequately phrased lemma, allowing new implementation 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 An example of proof-producing extraction

We begin by illustrating the compilation process on the example Fiat program from section 2. We synthesize a Facade program p according to the following specification[5], which we summarize as $\boxed{args} \overset{\texttt{p}}{\underset{\varnothing}{\rightsquigarrow}} [\![\texttt{"out"} \mapsto \underline{\text{p}}]\!] :: \boxed{args}$:

- p, when started in an initial state containing the arguments $\$author$ and $\$books$, must be safe (it must not violate function preconditions, access undefined variables, leak memory, etc.).
- p, when started in a proper initial state, must reach (if it terminates) a state where the variable "out" has one of the values allowed by the nondeterministic program $\underline{\text{p}}$ shown above.

Replacing $\underline{\text{p}}$ with our example, we need to find a program p such that

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

We use our first *compilation lemma* (with a few examples shown in Fig. 2) to connect the semantics of Fiat's bind operation (the $\leftarrow$ operator of monads [27]) to the meaning of $\rightsquigarrow$, which yields the following synthesis goal:

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

[5] In the following, underlined variables such as $\underline{\text{comp}}$ are Fiat computations, and italicized variables such as r are Gallina variables.

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

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

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

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

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

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

```
pinit ;
end = List.empty?(ls);
While (not end)
  hd = List.pop!(ls); pbody ; end = List.empty?(ls);
EndWhile;
List.delete!(ls);
```

(d) The Facade `LOOP(pinit, pbody, ls)` macro.

Fig. 2: A few rules of our synthesizing compiler.

In this step, we have broken down the assignment to `"out"` of a Fiat-level bind ($\mathtt{rows} \leftarrow \ldots; \ldots$) into the assignment of two variables: `"tmp"` to the intermediate list of authors, and `"out"` to the final result. The :: operator separates entries in a list of bindings of Facade variables to nondeterministic Fiat terms. The ordering of the individual bindings matters: the Fiat term that we assign to `"out"` depends on the particular value chosen for `"tmp"`, bound locally as r.

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 $\boxed{\boxed{args}}$) 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.

$$\boxed{\boxed{args}} \overset{\mathsf{p}_1}{\underset{\varnothing}{\rightsquigarrow}} [\![\texttt{"tmp"} \mapsto \underline{\mathsf{shuffle}}(\$books \cap \ldots) \text{ as } r]\!] :: \boxed{\boxed{args}}$$

$$\begin{array}{l}[\![\texttt{"tmp"} \mapsto \underline{\mathsf{shuffle}}(\$books \cap \ldots) \text{ as } r]\!] \\ :: \boxed{\boxed{args}}\end{array} \overset{\mathsf{p}_2}{\underset{\varnothing}{\rightsquigarrow}} \begin{array}{l}[\![\texttt{"tmp"} \mapsto \underline{\mathsf{shuffle}}(\$books \cap \ldots) \text{ as } r]\!] :: \\ [\![\texttt{"out"} \mapsto \mathtt{ret}\ (\mathtt{map}\ (\lambda\, r \Rightarrow r.\mathtt{Title})\ r)]\!] :: \boxed{\boxed{args}}\end{array}$$

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 = BTree.find($books, $author)` is a good choice for p_1 (this is the *call rule* for `BTree.find`). We are therefore only left with p_2 to synthesize: noticing the common prefix of the starting and ending states, we apply a rule (called *chomp* in our development) allowing us to set aside the common prefix and focus on the tail of the pre- and post-states, transforming the problem into

$$\begin{array}{l}\forall\, r.\ r \in \underline{\mathsf{shuffle}}(\$books \cap \{\mathtt{b} \mid \mathtt{b.by} = \$author\}) \implies \\ \qquad \boxed{\boxed{args}} \overset{\mathtt{p_2}}{\underset{[\texttt{"tmp"} \mapsto r]}{\rightsquigarrow}} [\![\texttt{"out"} \mapsto \mathtt{ret}\ (\mathtt{map}\ (\lambda\, r \Rightarrow r.\mathtt{Title})\ r)]\!] :: \boxed{\boxed{args}}\end{array}$$

The additional mapping pictured under the $\rightsquigarrow$ arrow indicates that the initial and final states must both map `"tmp"` to the same value r. In this form, we can first rewrite `map` to $\mathtt{fold_L}$, at which point the synthesis goal matches the conclusion of the $fold_L$ rule shown in Fig. 2c: 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 $\mathtt{LOOP(p_{init}, p_{body}, rows)}$ 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: from Fiat to Facade

4.1 The Facade language

We start with a brief description of our newly designed target language, Facade. Facade is an Algol-like untyped imperative language operating on Facade states, which are finite maps from variable names to Facade values (either scalars, or nonnull values of heap-allocated ADTs). Syntactically, Facade includes standard programming constructs like assignments, conditionals, loops, function calls, and recursion. What distinguishes the language is its operational semantics, pictured partially in Fig. 3. First, that semantics follows that of Cito in supporting modularity by modeling calls to externally defined functions via preconditions and postconditions. 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) to simplify 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.

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 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

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

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

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

Fig. 3: Selected syntax & operational semantics of Facade [28].

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 names to their axiomatic specifications. The semantics is nondeterministic in the sense that there can be more than one possible st'.

Modularity is achieved through the CallAx rule, allowing a Facade program to call a function via its specification in Ψ. A function call produces a return value r and a list of output values $\overline{v}$ representing the result of in-place modification of input ADT arguments $\overline{\texttt{y}}$. 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 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 (possibly deallocating them).

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. Its soundness proof has the flavor of justifying a new type system for an existing language, since Facade's syntax matches that of Cito rather closely.

4.2 Fiat and Facade states

We connect Fiat's semantics to those of Facade by introducing a notion of *Fiat states*, which allow us to express pre and post-conditions in a concise and homogeneous way, facilitating syntax-driven compilation. Each Fiat state (denoted as st) describes a set of Facade states (denoted as st): in Facade, machine states are unordered collections of names and values. Fiat states, on the other hand, are ordered collections of bindings (sometimes called *telescopes*), each containing a variable name and a set of permissible values for that variable.

$$\frac{\begin{array}{c}\forall\, \texttt{k},\ \texttt{v}.\ \texttt{st(k)} = \texttt{Scalar}(v) \rightarrow \textit{ext}\,(\texttt{k}) = \texttt{Scalar}(v) \\ \forall\, \texttt{k},\ \texttt{v}.\ \texttt{st(k)} = \texttt{ADT}(v) \leftrightarrow \textit{ext}\,(\texttt{k}) = \texttt{ADT}(v)\end{array}}{\texttt{st} \lesssim \varnothing \uplus \textit{ext}}\ \textsc{EqvStNil}$$

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

Fig. 4: Equivalence relation on Fiat and Facade states. Because Facade does not allow us to leak ADT values, we require that all bindings pointing to ADT values in `st` be reflected in $\textit{st} \uplus \textit{ext}$, and vice versa. For scalars, we only require that bindings in $\textit{st} \uplus \textit{ext}$ be present in `st`.

For example, the telescope $[\![\texttt{"x"} \mapsto \{x \mid x > 0\}\ \texttt{as}\ x]\!] :: [\![\texttt{"y"} \mapsto \texttt{ret}\ (x,\ x + 1)]\!]$ describes all machine states in which `"x"` maps to a positive value x and `"y"` maps to the pair $(x,\ x + 1)$. Each variable in a Fiat state is annotated with a function `wrap` describing how to inject values of its type in and out of the concrete type used at the Facade level (e.g. a linked list may be extracted to a vector, as in our example).

Finally, to be able to implement the aforementioned *chomp* rule, Fiat states are extended with an unordered map ($\textit{ext}$) from names to concrete values. A full Fiat state is thus composed of a telescope $\textit{st}$ and an extra collection of bindings $\textit{ext}$, written $\textit{st} \uplus \textit{ext}$. We relate Fiat states to Facade states using the ternary predicate $\texttt{st} \lesssim \textit{st} \uplus \textit{ext}$ defined in Fig. 4, which ensures that the values assigned to variables in the Facade state `st` are compatible with the bindings described in the Fiat state $\textit{st} \uplus \textit{ext}$.

4.3 Proof-generating extraction by synthesis

Armed with this predicate, we are ready for the full definition of $\textit{st} \overset{\texttt{p}}{\underset{\textit{ext}}{\rightsquigarrow}} \textit{st'}$:

- $\forall\, \texttt{st}.\ \texttt{st} \lesssim \textit{st} \uplus \textit{ext} \implies (\texttt{p}, \texttt{st})\!\downarrow$
  For any initial Facade state `st`, if `st` is in relation with the Fiat state $\textit{st}$ extended by $\textit{ext}$, then it is safe to run the Facade program `p` from state `st`.
- $\forall\, \texttt{st}, \texttt{st'}.\ \texttt{st} \lesssim \textit{st} \uplus \textit{ext} \wedge (\texttt{p}, \texttt{st}) \Downarrow \texttt{st'} \implies \texttt{st'} \lesssim \textit{st'} \uplus \textit{ext}$
  For all initial and final Facade states `st` and `st'`, if `st` is in relation with the Fiat state $\textit{st}$ extended by $\textit{ext}$, 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 $\textit{st'}$ extended by $\textit{ext}$.

This definition is enough to concisely and precisely phrase the three types of lemmas required to synthesize Facade programs: properties of the $\rightsquigarrow$ relation used to drive the proof search and provide the extraction architecture; connections between the $\rightsquigarrow$ relation and Fiat's semantics, used to reduce extraction of Fiat programs to that of Gallina programs; and connections between Fiat and Facade, such as the FoldL rule of Fig. 2c (users provide additional lemmas of the latter kind to extend the scope of the compiler and broaden the range of source programs that the 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 $\underline{\mathsf{f}}\ \boldsymbol{x}_1 \ldots \boldsymbol{x}_n$ mixing Gallina code with calls to external ADTs, we generate a specification $\rceil\underline{\mathsf{f}}\lceil$ based on the $\rightsquigarrow$ predicate (which itself is defined in terms of Facade's operational semantics):

$$\rceil\underline{\mathsf{f}}\lceil \triangleq \exists\, \mathtt{p}.\ \forall\, \boldsymbol{x}_1 \ldots \boldsymbol{x}_n.\, [\![\texttt{"x}_1\texttt{"} \mapsto \mathtt{ret}\ \boldsymbol{x}_1]\!] :: \ldots :: [\![\texttt{"x}_n\texttt{"} \mapsto \mathtt{ret}\ \boldsymbol{x}_n]\!] \underset{\varnothing}{\overset{\mathtt{p}}{\rightsquigarrow}} [\![\texttt{"out"} \mapsto \underline{\mathsf{f}}\ \boldsymbol{x}_1 \ldots \boldsymbol{x}_n]\!] \qquad (1)$$

From this starting point, extraction proceeds by analyzing the shapes of the pre- and post-states to determine applicable compilation rules, which are then used to build a Facade program progressively. This stage explains why we chose strongly constrained representations for pre and post-states: where typical verification tasks compute verification conditions from the program's source, we compute the program from carefully formulated pre- and postconditions (proper care in designing the compilation rules and their preconditions obviates the need for backtracking).

In practice, this syntax-driven process is implemented by a collection of matching functions written in Ltac. These may either fail, or solve the current goal by applying a lemma, or produce a new goal by applying a compilation lemma of the form shown in Fig. 2. Our extraction architecture is extensible: the main loop exposes hooks that users can rebind to call their own matching rules. Examples of such rules are provided in section 6.1. Our focus is on extracting efficient code from Gallina EDSLs, so the set of rules is tailored to each domain and does not cover all possible programs (in particular, we do not have support for arbitrary fixpoints or pattern-matching constructs; we use custom lemmas mapping specific matches to specific code snippets or external functions). When the compiler encounters an unsupported construct $\underline{\mathsf{C}}$, it stops and presents the user with a goal of the form $\boldsymbol{pre} \underset{\boldsymbol{ext}}{\overset{?}{\rightsquigarrow}} [\![\mathsf{k} \mapsto \underline{\mathsf{C}}]\!] :: \boldsymbol{post}$, indicating which piece is missing so the user can provide the missing lemmas and tactics.

In our experience, debugging proof search and adding support for new constructs is relatively easy, though it does require sufficient familiarity with Coq. Typically, our compiler would have two classes of users: *library developers*, who interactively implement support for new DSLs (developing compilation tactics requires manual labor similar to writing a domain-specific compiler); and *final users*, who write programs within supported DSLs and use fully automated compilation tactics.

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 Fig. 5a), 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 Fig. 5b). Each method of this implementation is assigned an operational specification $\upharpoonleft\underline{\mathsf{m}}_{\mathtt{impl}}\upharpoonright$ (equation 1), from which we extract (using proof-producing synthesis, optionally augmented with user-specified lemmas and tactics) a verified Facade implementation $\mathtt{m}_{\mathtt{impl}}$ (section 4.3) that calls into a number of external functions (Ψ, Fig. 3), as shown in Fig. 5c.

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 Fig. 3): for each high-level specification $\underline{\mathsf{meth}}_{\mathsf{spec}}$, we export the following (written $\lceil\underline{\mathsf{meth}}_{\mathsf{spec}}\rceil$):

$$\begin{aligned}
&\mathtt{Pre} \triangleq \lambda\ \mathit{args} \Rightarrow \exists\ r_S\ r_I\ xs.\ r_S \approx r_I \wedge \mathit{args} = [r_I] \mathbin{+\!\!+} xs\\
&\mathtt{Post} \triangleq \lambda\ \mathit{args} \Rightarrow \exists\ r_S\ r_I\ r'_S\ r'_I\ v\ xs.\\
&\quad r_S \approx r_I \wedge (r'_S,\ v) \in \underline{\mathsf{meth}}_{\mathsf{spec}}\ r_S\ xs \wedge\\
&\quad r'_S \approx r'_I \wedge \mathit{args} = [(r_I,\ r'_I)] \mathbin{+\!\!+} (\mathtt{zip}\ xs\ xs)
\end{aligned}$$

Since we are working in an object-oriented style at the high level, our low-level code follows a convention of an extra “self” argument added to each method, written in this logical formulation as r_S for spec-level “self” values and r_I for implementation-level “self” values.

A *generic* proof guarantees that the operational specifications $\upharpoonleft\underline{\mathsf{meth}}_{\mathtt{impl}}\upharpoonright$ used to synthesize Facade code indeed refine the axiomatic specifications $\lceil\underline{\mathsf{meth}}_{\mathsf{spec}}\rceil$ exported by our Facade module. Compiling this Facade module via our new formally verified Facade compiler produces a correct Bedrock module, completing Theorem 1:

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

The final Bedrock module satisfies the *original, high-level* Fiat specifications. It specifies its external dependencies Ψ, for which verified assembly implementations must be provided as part of the final *linking* phase, which happens entirely inside of Coq. After linking, we obtain a closed, executable Bedrock module, exposing an axiomatic specification directly derived from the original, high-level ADT specification. Our implementation links against verified hand-written implementations of low-level indexing structures, though it would be possible to use the output of any compiler emitting Bedrock assembly code.

6 Evaluation

6.1 Microbenchmarks

We first evaluated our pipeline by extracting a collection of twenty six Gallina programs manipulating machine words, lists, and nested lists, with optional non-deterministic choices. Extraction takes a few seconds for each program, ranging 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. All examples, and the corresponding outputs, are included in a literate Coq file available online. These examples illustrate that our extraction engine supports a fluid, extensible source language, including subsets of Gallina and many nondeterministic Fiat programs.

```
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.Procs,

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

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

(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 (Get 0) (snd a))

GetCPUTime
a ← bfind r (_, $id, _);
ret (fst a, revmap (Get 2) (snd a))
```

(b) The output of Fiat, after refining the specifications presented in Fig. 5a. 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);
  head' = Tuple.get(head, 0);
  Call Tuple.delete!(head, 3);
  Call List[W32].push(ret, head');
  test = List[Tuple].empty?(procs)
EndWhile;
Call List[Tuple].delete!(procs);
```

(c) Facade output 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).

Fig. 5: Different stages of a process-scheduler compilation example (see also the annotated 'ProcessScheduler.v' file).

6.2 Relational queries

To evaluate our full pipeline in realistic conditions, we targeted the query-structure ADT library of the Fiat paper [4] as well as an ADT modeling process scheduling inspired by Hawkins et al [7]. This benchmark starts from high-level Fiat specifications (as shown in Fig. 5a) and outputs a closed Bedrock module, linked against a hand-verified nested-binary-tree implementation.

From Fiat specifications we derive a collection of nondeterministic Fiat programs (one per ADT method, as demonstrated in Fig. 5b), then extract each method to Facade Fig. 5c) and compile to Bedrock. Extraction is fully automatic; it draws from the default pool of extraction lemmas (about conditionals, constants, arithmetic operations, etc.) and from bag-specific lemmas that we added to 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).

Fig. 6 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"). 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. 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).

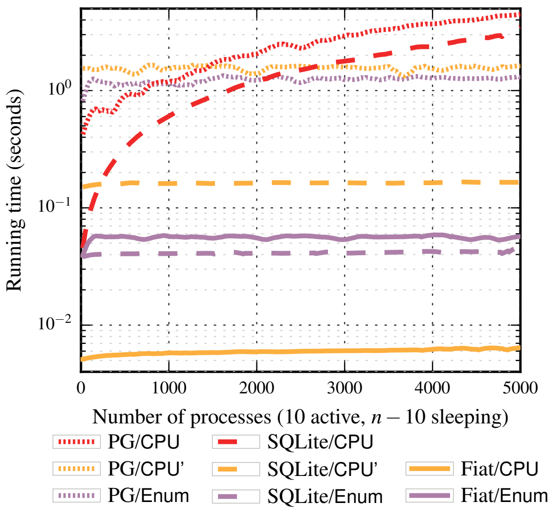

Fig. 6: Process scheduler benchmarks.

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: without an explicit "`$state IN (0, 1)`" clause, both database management systems missed the skip-scan opportunity and exhibited asymptotically suboptimal linear-time behavior, so we had to tweak the queries fed to PostgreSQL and SQLite to obtain good GetCPUTime performance (in contrast, the optimizer in our system can be guided explicitly by adding compiler hints in the form of extra tactics, without modifying the specifications).

Of course, our implementation does much less work than a database engine; the strength of our approach is to expose an SQL-style interface while enabling generation of specialized data-structure-manipulation code, allowing programmers to benefit from the conciseness and clarity of high-level specifications without incurring the overheads of a full-fledged DBMS.

Trusted base Our derivation assumes ensemble extensionality and Axiom K. Our trusted base comprises the Coq 8.4 checker [25] (∼10 000 lines of OCaml code), the semantics of the Bedrock IL and the translator from it to x86 assembly (∼1200 lines of Gallina code), an assembler, and wrappers for extracted methods (∼50 lines of x86 assembly). We used Proof General [2] for development.

7 Related work

Closely related to our work is a project by Lammich [10] that uses Isabelle/HOL to refine functional programs to an embedded imperative language that requires garbage collection. This approach has been applied to various complex algorithms, whereas our focus is on fully automatic derivation from highly regular specs. Both approaches use some form of linearity checking to bridge the functional-imperative gap (Lammich et al. use separation logic [20] and axiomatic semantics, while we apply Facade's lighter-weight approach: decidable syntactic checks applied after-the-fact, with no explicit pointer reasoning). A recent extension [11] targets LLVM directly. Crucially, the initial work only targets Imperative/HOL and its extension does not support linking against separately verified libraries, while our pipeline allows linking, inside of Coq, low-level programs against verified libraries written in any language of the Bedrock ecosystem. Finally, we have 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 Kumar et al. [17,8] focuses on extracting terms written in a purely functional subset of HOL4's logic into the CakeML dialect of ML. The main differences with our pipeline are optimization opportunities, extensibility, and external linking. Indeed, while the compiler to CakeML 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.

Recent work by Protzenko et al. [19] achieves one of our stated goals (efficient extraction to low-level code, here from F* to C) but does not provide formal correctness guarantees for the extracted code (the tool, KreMLin, consists of over 15,000 lines of unverified OCaml code). Additionally, KreMLin requires source programs to be written in a style matching that of the extracted code: instead of extending the compiler with domain-specific representation choices and optimizations, users must restrict their programs to the Low* subset of F*.

One last related project is the compiler of the Cogent language [18]. Its sources are very close to Facade's (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 differ 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. Cogent uses a linear type system to establish memory safety, while we favor extensibility over completeness, relying on lemmas to justify the compilation of arbitrary Gallina constructs.

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 [14], but the corresponding concrete implementation itself is unverified. The recent development of CertiCoq [1], a verified compiler for Gallina, has significantly improved matters over unverified extraction, but it only supports pure Gallina programs, and it uses a fixed compilation strategy. In contrast, our pipeline ensures that nondeterministic specifications are preserved down to the generated Bedrock code and grants user fine control over the compilation process.

Compiler verification Our compilation strategy allows Fiat programs to depend on separately compiled libraries. This contrasts with verified compilers like CakeML [9] or CompCert [13]: 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 [23] 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 [21], 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, paired with verified checkers 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 [26], do not focus on verification. Others, like Rhodium [12], 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 [24], which lets CompCert users soundly describe program transformations using an EDSL. Our approach is similar insofar as we assemble DSL 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 [15]. We use the syntactic structure of our specialized pre- and postconditions to drive synthesis: the idea of strongly constraining the search space is inherited from the syntax-guided approach pioneered in the *Sketch* language [22]. 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 [16] or C [6]) 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 extraction technique presented in this paper is a convenient and lightweight approach for generating certified extracted programs, reducing the trusted base of verified programs to little beyond a 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 linking 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 implemented, compiled, and verified using arbitrary languages and tools.

Acknowledgments. This work has been supported in part by NSF grants CCF-1512611 and CCF-1521584, and by DARPA under agreement number FA8750-16-C-0007. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq. In: CoqPL'17: The Third International Workshop on Coq for PL (Jan 2017)
2. Aspinall, D.: Proof General: A generic tool for proof development. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000. pp. 38–42 (2000). https://doi.org/10.1007/3-540-46419-0_3
3. Chlipala, A.: The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In: ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA - September 25 - 27, 2013. pp. 391–402 (2013). https://doi.org/10.1145/2500365.2500592
4. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 689–700 (2015). https://doi.org/10.1145/2676726.2677006
5. Dijkstra, E.W.: A constructive approach to the problem of program correctness (Aug 1967), https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD209.PDF, circulated privately
6. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: International Conference on Interactive Theorem Proving, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. pp. 99–115 (2012). https://doi.org/10.1007/978-3-642-32347-8_8
7. Hawkins, P., Aiken, A., Fisher, K., Rinard, M.C., Sagiv, M.: Data representation synthesis. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 38–49 (2011). https://doi.org/10.1145/1993498.1993504
8. Ho, S., Abrahamsson, O., Kumar, R., Myreen, M.O., Tan, Y.K., Norrish, M.: Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. Lecture Notes in Computer Science p. 646–662 (2018). https://doi.org/10.1007/978-3-319-94205-6_42
9. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: Cakeml: A verified implementation of ML. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, January 20-21, 2014. pp. 179–192 (2014). https://doi.org/10.1145/2535838.2535841
10. Lammich, P.: Refinement to Imperative/HOL. In: International Conference on Interactive Theorem Proving, ITP 2015, Nanjing, China, August 24-27, 2015. pp. 253–269 (2015). https://doi.org/10.1007/978-3-319-22102-1_17
11. Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA. pp. 22:1–22:19 (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.22
12. Lerner, S., Millstein, T.D., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. pp. 364–377 (2005). https://doi.org/10.1145/1040305.1040335

13. Leroy, X.: Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006. pp. 42–54 (2006). https://doi.org/10.1145/1111037.1111042
14. Letouzey, P.: A new extraction for Coq. In: International Workshop on Types for Proofs and Programs, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002. pp. 200–219 (2002). https://doi.org/10.1007/3-540-39185-1_12
15. Manna, Z., Waldinger, R.J.: A deductive approach to program synthesis. ACM Transactions on Programming Languages and Systems **2**(1), 90–121 (Jan 1980). https://doi.org/10.1145/357084.357090
16. Myreen, M.O., Gordon, M.J.C., Slind, K.: Decompilation into logic - improved. In: Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012. pp. 78–81 (2012), https://ieeexplore.ieee.org/document/6462558/
17. Myreen, M.O., Owens, S.: Proof-producing synthesis of ML from higher-order logic. In: ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, Copenhagen, Denmark, September 9-15, 2012. pp. 115–126 (2012). https://doi.org/10.1145/2364527.2364545
18. O'Connor, L., Rizkallah, C., Chen, Z., Amani, S., Lim, J., Nagashima, Y., Sewell, T., Hixon, A., Keller, G., Murray, T.C., Klein, G.: COGENT: certified compilation for a functional systems language. CoRR **abs/1601.05520** (2016), https://arxiv.org/abs/1601.05520
19. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Béguelin, S.Z., Delignat-Lavaud, A., Hritcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified low-level programming embedded in F*. Proceedings of the ACM on Programming Languages **1**(ICFP), 17:1–17:29 (2017). https://doi.org/10.1145/3110261
20. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: IEEE Symposium on Logic in Computer Science, LICS 2002, 22-25 July 2002, Copenhagen, Denmark. pp. 55–74 (2002). https://doi.org/10.1109/LICS.2002.1029817
21. Sewell, T.A.L., Myreen, M.O., Klein, G.: Translation validation for a verified OS kernel. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, June 16-19, 2013. pp. 471–482 (2013). https://doi.org/10.1145/2491956.2462183
22. Solar-Lezama, A.: The sketching approach to program synthesis. In: Asian Symposium on Programming Languages and Systems, APLAS 2009, Seoul, Korea, December 14-16, 2009. pp. 4–13 (2009). https://doi.org/10.1007/978-3-642-10672-9_3
23. Stewart, G., Beringer, L., Cuellar, S., Appel, A.W.: Compositional CompCert. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 275–287 (2015). https://doi.org/10.1145/2676726.2676985
24. Tatlock, Z., Lerner, S.: Bringing extensibility to verified compilers. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010. pp. 111–121 (2010). https://doi.org/10.1145/1806596.1806611
25. The Coq Development Team: The Coq Proof Assistant Reference Manual (2012), https://coq.inria.fr, version 8.4
26. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages as libraries. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 132–141 (2011). https://doi.org/10.1145/1993498.1993514
27. Wadler, P.: Comprehending monads. Mathematical Structures in Computer Science **2**(4), 461–493 (1992). https://doi.org/10.1017/S0960129500001560

28. Wang, P.: The Facade language. Tech. rep., MIT CSAIL (2016), https://people.csail.mit.edu/wangpeng/facade-tr.pdf
29. Wang, P., Cuellar, S., Chlipala, A.: Compiler verification meets cross-language linking via data abstraction. In: ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014. pp. 675–690 (2014). https://doi.org/10.1145/2660193.2660201