Compilation Using Correct-by-Construction Program Synthesis

Abstract

Extracting and compiling certified programs may introduce bugs in otherwise proven-correct code, reducing the extent of the guarantees that proof assistants and correct-by-construction program-derivation frameworks provide. We present a novel approach to the extraction and compilation of embedded domain-specific languages developed in a proof assistant (Coq), showing how it allows us to extend correctness guarantees all the way to a verification-aware assembly language. Our core idea is to phrase compilation of shallowly embedded programs to a lower-level deeply embedded language as a synthesis problem, solved using simple proof-search techniques. This technique is extensible (support for individual language constructs is provided by a user-extensible database of compilation tactics and lemmas) and allows the source programs to depend on axiomatically specified methods on externally implemented data structures, delaying linking to the assembly stage. We do proof-generating static analysis of object lifetimes to avoid the need for a garbage collector, so that our output code is suitable for embedded systems or system infrastructure. We have composed our new transformation with others in Coq to provide the first fully proof-generating automatic translation from SQL-style relational programs into executable assembly code.

1. Introduction

The Fiat program derivation framework of Delaware et al. automatically 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. For example, Fiat's query structures library refines the following query into a correct-by-construction program similar to the following one:

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

```
ret foldl (λ count book.
             count + bcount orders book.isbn)
    (bfind books $author)
```

In addition, Fiat programs can be parametrized by abstract data types (ADTs), whose behavior is axiomatically specified and whose implementation is left unspecified. 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 *book*.isbn.

To compile the results of Fiat's synthesis into executable binaries, users *extract* the refined programs to other languages such as OCaml. 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. 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: bugs in the extraction mechanism or in the compiler may cause the binaries to misbehave.

This paper pushes the guarantees offered by a correct-by-construction program synthesis framework embedded in a proof assistant (Fiat) all the way down to a verification-aware assembly language (Bedrock [1]). Our main contribution is a translation from shallowly embedded refined Fiat programs to deeply embedded programs written in Facade, a C++-inspired imperative language. Rather than write a traditional compiler, we phrase the problem as a *synthesis task* happening inside the proof assistant: instead of relying on extraction, we propose to consider Fiat programs themselves as specifications expressed in a functional language, and to use them to drive a correct-by-construction synthesis procedure in sequent-calculus style.

Our main contributions are to introduce a novel, extensible approach to the sound compilation of shallowly embedded domain-specific languages in a proof assistant using the

source language as a specification driving a synthesis process, and to show how this approach accommodates dependencies on external axiomatically specified operations, allowing us to link against external libraries written in other languages. Another notable property of our translation is that it supports nondeterministic source programs; for instance, querying a data table for all matching rows ought to be able to return the rows in any order, if the user has no particular sorting requirement in mind. Different data structures may imply different natural orderings that are efficient to produce.

The work we present here provides the last step in demonstrating the feasibility of a a mechanically certified automatic translation pipeline from declarative specifications to assembly language with support for abstraction of externally verified data structure implementations.

Our approach has many advantages beyond suppressing the trust dependency on potentially unsound extraction and compilation procedures:

- It is lightweight: it does not require reflecting the entirety of Gallina into a deeply embedded language before compiling
- It is modular: each part of the compilation logic is expressed as a derivation rule, proved as a Coq theorem; support for particular constructs can be provided by adding compilation lemmas, extending the supported source language. Similarly, support for domain-specific optimizations is easy to add, enabling case-specific performance improvements over generic extraction.
- It allows us to link against axiomatically specified libraries, with external calls expressed using Fiat programs that have not been fully refined into deterministic code, as in the following example:

  $x \leftarrow$ **rand**();
  ret $(x + x)$

- It compiles directly to a relatively bare language, with fully manual memory management, providing performance and predictability advantages[1].

To demonstrate the practical applicability of this approach, we evaluated our extraction routines on two set of benchmarks:

- A collection of micro-benchmarks, starting from small hand-written Fiat programs manipulating variables, conditions, and lists of machine words and generating Facade programs automatically.
- A collection of larger benchmarks, demonstrating the capabilities and performance of extraction on real-life programs similar to those of the original Fiat paper. These benchmarks start from high-level specifications of database queries: these queries are refined down to functional programs parametrized by external ADTs implementing axiomatically specified nondeterministic bag operations, then extracted to Facade, and compiled down to Bedrock.

[1] Beyond raw performance, the absence of a garbage collector makes it easier to get real-time guarantees, and to reason about potential side-channel attacks on sensitive code.

2. Background

Thanks to advances of compiler-verification research, strengthening the trust chain of verified programs to exclude the dependency on the compiler is now relatively easy: verified compilers are available for a variety of languages, including subsets of C or C++ such as CompCert [10] and Cito [24] as well as variants of ML such as CakeML [7]. Unfortunately, no mechanism, let alone a certified one, exists to extract Gallina code to one of these languages.

Achieving sound extraction, on the other hand, is much harder, as the traditional technique for writing verified compilers does not directly apply: instead of writing and verifying a function translating deeply embedded abstract syntax trees from one language to another, the authors of a sound extraction procedure must start with a shallowly embedded program and produce a deeply embedded term encoded using an inductive data type.

Phrasing the extraction problem in compiler verification terms would therefore require primitives for *quoting* terms, allowing one to reason about the semantics and structure of Gallina terms in Gallina itself; unfortunately, Gallina does not offer such facilities. Instead, one must produce a *verifying* compiler; that is, a compilation procedure written in a different language, which inspects Gallina terms in a possibly unsound way and produces not only the deeply embedded AST of a compiled program, but also a proof that this derivation is correct. Thus the extraction mechanism itself is not verified, but each extraction is accompanied by a Coq-checked proof that guarantees its correctness.

One systematic alternative is to reflect the entirety of Gallina as a Gallina inductive type. In this approach, the extraction process consists of two steps: the first step, implemented in Coq's Turing-complete language for inspecting and manipulating Gallina terms (Ltac), inspects the AST of Gallina terms to mirror them into deeply embedded programs, producing a proof that connects these programs to the original terms. The second step, written and verified in Gallina, uses typical verified compilation techniques to compile this deep embedding of Gallina into the target language. This approach, although rather heavy (it requires writing a specification of the entirety of Gallina inside of Gallina itself), would be practical if our aim were to compile self-contained general Gallina programs; our focus, however, is not on such programs: it is on Fiat programs, a specific class of Gallina terms sharing common properties that make them amenable to more specialized compilation, supporting linking against external code and manual memory management. In addition, our starting points are terms that may use a nondeterministic choice

operator, and that feature may not be translated to executable code in a generic way. Rather, we depend on the code we link against to resolve the nondeterminism.

3. Technical outline

3.1 Example of compilation by synthesis

We begin by illustrating the compilation process on a simple example, starting with the following Fiat program p:

$p \coloneqq r \leftarrow \textbf{rand}()$
$\quad \textbf{ret}\ (\mathit{if}\ r \prec 0\ \mathit{then}\ -r\ \mathit{else}\ r)$

This program samples a random number using a call to an external function and returns its absolute value ($\prec$ denotes a Boolean version of the less-than predicate). To compile this program to Facade (described later), we synthesize a Facade program `p`, according to the following specification[2]:

- `p`, when started in a blank state (no variables defined, written $\varnothing$) must be safe; that is, `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 blank 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 a value equivalent to the Gallina term p shown above.

We write $\varnothing \overset{\texttt{p}}{\rightsquigarrow} [\texttt{"out"} \leftarrow p]$ to summarize this specification; we read it as "the Facade program `p`, starting in a blank state, behaves properly and stores a value equivalent to the Fiat term 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[3]. The precise definition shown in subsection 3.2 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 looks like the following:

```
tmp := Call rand()
comp := tmp < 0
If comp Then
  out := -tmp
Else
  out := tmp
EndIf
```

Expanding the definition of p, we find that we need to find a program `p` such that

$$\varnothing \overset{\texttt{p}}{\rightsquigarrow} \left[\texttt{"out"} \leftarrow \begin{array}{l} r \leftarrow \textbf{rand}() \\ \textbf{ret}\ (\mathit{if}\ r \prec 0\ \mathit{then}\ -r\ \mathit{else}\ r) \end{array}\right]$$

We use our first *compilation lemma* to connect the semantics of Fiat's bind operation (the $\leftarrow$ operator of monads [23]) to the meaning of $\rightsquigarrow$, which yields the following synthesis goal:

$$\varnothing \overset{\texttt{p}}{\rightsquigarrow} [\texttt{"tmp"} \leftarrow \textbf{rand}()\ \textbf{as}\ r] :: \\ [\texttt{"out"} \leftarrow \textbf{ret}\ (\mathit{if}\ r \prec 0\ \mathit{then}\ -r\ \mathit{else}\ r)]$$

In this step, we have broken down the assignment of a Fiat-level binding operation ($r \leftarrow \textbf{rand}(); \ldots$) to a single variable into the construction of two variables: `"tmp"`, holding the result of the call to **rand**(), and `"out"`, holding the final result[4]. 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 that the call to **rand**() returns.

We then break down `p` into two smaller programs: the first starts in a blank state and is only concerned with the assignment to `"tmp"`; the second one starts in a state where `"tmp"` is already assigned and uses that value to construct the final result:[5]

$$\varnothing \overset{\texttt{p1}}{\rightsquigarrow} [\texttt{"tmp"} \leftarrow \textbf{rand}()\ \textbf{as}\ r]$$

$$[\texttt{"tmp"} \leftarrow \textbf{rand}()\ \textbf{as}\ r] \overset{\texttt{p2}}{\rightsquigarrow} [\texttt{"tmp"} \leftarrow \textbf{rand}()\ \textbf{as}\ r] :: \\ [\texttt{"out"} \leftarrow \textbf{ret}\ \ldots]$$

At this point, a lemma about Facade's semantics tells us that `tmp := Call rand()` is a good choice[6] for `p1` (this is the *call rule* for **rand**). We are therefore only left with

[2] In the following, typewriter variables such as `prog` live in Facade land, Roman script variables such as comp are Fiat computations, and slanted variables such as p are Gallina terms.

[3] 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; putting additional constraints helps us avoid vacuous preconditions ($\top$) and unrealizable postconditions ($\bot$), and helps the compiler synthesize the Facade program.

[4] The double-colon operator (::) is syntactic sugar for a consing operation. In more detail, this notation describes states where the first variable `"tmp"` can map to any of the variables allowed by the call to random, and where the second variable maps to any of the values allowed by the addition operation: although the addition is deterministic, it depends on the value returned by random, bound as r at the Gallina level.

[5] Notice the similarity between the specification of `p1` and the original specification of `p`: just like `p` moved from a blank state to a state binding `"out"` to a Fiat expression, `p1` moves from a blank state to a state binding `"tmp"` to a Fiat expression.

[6] We use the same name to denote the specification of the **rand**() function as used by Fiat and the Facade-level specification of that same function. A user-provided lemma, written by the author of the **rand** function, connects these two meanings.

p2 to synthesize: noticing the common prefix of the starting and ending states, we apply the *chomp rule*, transforming the problem into

$$\forall r.\ \textbf{rand}() \rightsquigarrow r \implies$$
$$\varnothing \overset{\texttt{p2}}{\rightsquigarrow} [\texttt{"out"} \leftarrow \textbf{ret}\ (\textit{if}\ r \prec 0\ \textit{then}\ -r\ \textit{else}\ r)]\ [\texttt{"tmp"} \leftarrow r]$$

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. Since this is wrapped in a universal quantifier, what we are really requiring is that the synthesized program be valid regardless of the particular value returned by the call to **rand**[7]. In this form, after applying Coq's *Lam* rule, the synthesis goal matches the conclusion of the IF compilation lemma: given three programs which respectively set a comparison variable `"cmp"` to $r \prec 0$ (`pTest`), compile the true branch (`pTrue`) of the conditional, and compile the false branch of the conditional (`pFalse`), the Facade program `pTest; If cmp Then pTrue Else pFalse` obeys the specification above. This gives us three new synthesis goals, which we can handle recursively.

3.2 Technical details

The compilation procedure outlined above is implemented in Ltac.

3.2.1 The Facade Language

Facade [8] is an Algol-like untyped imperative language operating on Facade states, which are finite maps from variable names to Facade values. A Facade value can be either an integer (stack-allocated), which we call a "scalar", or it can be 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. 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, we have built *linearity* in Facade's operational semantics, enforcing that every ADT value on the heap will be referred to by exactly one live variable (no aliasing and no leakage). Such a pattern of linearity simplifies reasoning about the formal connection to functional programs: if every object has at most one referrent, then we can almost pretend that variables hold abstract values instead of pointers to mutable objects, while remaining compatible with standard semantics of C-like languages and their verified compilers.

Facade's operational semantics are defined by two predicates, $(\texttt{p}, \texttt{st})\downarrow$ and $(\texttt{p}, \texttt{st}, \texttt{st'})\Downarrow$, expressing respectively that 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). 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 a rule allowing a Facade program to call a function specified axiomatically. A function call's effect includes both the return value and a list of "output values." These output values represent the result of in-place modification of input ADT arguments. When an input argument is an ADT, the corresponding output value can be either (1) another ADT representing the new value of the input argument or (2) a null value indicating that the input ADT 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 on input argument values to the callee. A postcondition is a predicate on input values, output values, and return value. The semantics prescribes that such a function call will nondeterministically pick a list of output values and a return value satisfying the postcondition, and use them to update the relevant variables in the caller's post-call 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 assignmemts, to avoid losing their current payloads and causing memory leaks. Restrictions of this flavor appear throughout the operational semantics.

We have implemented a verified compiler from Facade to Cito, and from there we reuse established infrastructure to connect into the Bedrock framework for verified assembly code. Our Facade-to-Cito compiler accepts a record containing (1) a Facade program, (2) an axiomatic specification for the functions that the program exports, (3) a list of name/axiomatic-spec pairs representing the available functions the program can call from other modules, and (4) a proof that the program actually "refines" (i.e. implements) the export spec. Through composition with the verified Cito compiler, the Facade compiler generates a Bedrock module with the given export spec (meaning the assembly code in the Bedrock module satisfies the export spec). The refinement in (4) is defined in a form morally equivalent to the $pre \overset{\texttt{p}}{\rightsquigarrow} post$ relation introduced in the previous section. Thus the Facade-to-Bedrock compiler is readily connected to the Fiat-to-Facade compiler described in this paper to form a complete pipeline.

[7] A slight subtlety here is that the formulation may lead one to think that we may end up constructing a different program for each value of r. This is not the case: `p2` is an existential variable whose context does not contain r; it cannot depend on it.

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

$$\frac{\forall v_0.\ v_0 \in v \implies t\ v_0 \overset{\mathtt{p}}{\underset{[\mathtt{k} \leftarrow v_0]::ext}{\rightsquigarrow}} t'\ v_0}{[\mathtt{k} \leftarrow v\ \mathbf{as}\ v_0] :: (t\ v_0) \overset{\mathtt{p}}{\underset{ext}{\rightsquigarrow}} [\mathtt{k} \leftarrow v\ \mathbf{as}\ v_0] :: (t'\ v_0)}\ \textsc{Chomp}$$

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

$$\frac{st \overset{\mathtt{p}}{\underset{ext}{\rightsquigarrow}} [\_ \leftarrow \mathsf{comp}\ \mathbf{as}\ x] :: [\mathtt{k} \leftarrow f\,x] :: st'}{st \overset{\mathtt{p}}{\underset{ext}{\rightsquigarrow}} \left[\mathtt{k} \leftarrow \begin{matrix} x \leftarrow \mathsf{comp} \\ f\,x \end{matrix}\right] :: st'}\ \textsc{Bind}$$

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

$$\frac{\begin{matrix} \varnothing \overset{\mathtt{p_C}}{\underset{ext}{\rightsquigarrow}} [\mathtt{cmp} \leftarrow p_C] \\ p_C \implies \varnothing \overset{\mathtt{p_T}}{\underset{[\mathtt{cmp} \leftarrow p_C]::ext}{\rightsquigarrow}} [\mathtt{k} \leftarrow \mathsf{p_T}] \qquad \neg p_C \implies \varnothing \overset{\mathtt{p_F}}{\underset{[\mathtt{cmp} \leftarrow p_C]::ext}{\rightsquigarrow}} [\mathtt{k} \leftarrow \mathsf{p_F}] \end{matrix}}{\begin{matrix} \mathtt{p_C;\ If\ cmp\ Then\ p_T\ Else\ p_F} \\ \varnothing \overset{}{\underset{ext}{\rightsquigarrow}} [\mathtt{k} \leftarrow \mathit{if}\ p_C\ \mathit{then}\ \mathsf{p_T}\ \mathit{else}\ \mathsf{p_F}] \end{matrix}}\ \textsc{If}$$

(c) The *if* rule: provided that the three intermediate programs $\mathtt{p_C}$, $\mathtt{p_T}$, $\mathtt{p_F}$ respectively evaluate the condition of the if, implement its true branch, and implement its right branch, we can connect the semantics of Facade's **`If`** statement to the Gallina-level *if*.

Figure 1: Three rules used by our synthesizing compiler: *chomp*, *bind*, and *if*

3.2.2 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 a having a map relating names to Facade values and a number of propositions describing the relations and dependencies between these bindings, we may have a single data structure tracking exactly the required information to allow for compilation to succeed.

On the surface, Fiat states are similar to Facade states, as they both describe a collection of bindings. The two, however, differ crucially in a number of ways: indeed, Facade states are unordered collections of bindings, with dependencies between bindings being expressed by additional propositions learned as one reasons about program transitions. Fiat states, on the other hand, are self-contained, dependently typed heterogeneous ordered lists of bindings (also called *telescopes*), 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, allowing later bindings to depend on earlier values.

One can think of them as Fiat computations, where the Bind constructor is extended to allow naming certain variables[9]. Finally, for convenience and to be able to implement the aforementioned *chomp* rule, Fiat states are extended with an unordered map from names to single values, which morally represents the Fiat variables for which a particular value has been picked among permissible values.

We relate Fiat states to Facade states using a recursively defined ternary predicate

$$\mathtt{st} \approxeq st \uplus ext$$

that ensures that the values assigned to each variable in the Facade state $\mathtt{st}$ are compatible with the bindings described in the Fiat state $st \uplus ext$, and that the values chosen for each of the Facade computations are compatible with choices made for previous bindings. Because of Facade's manual memory management model, we require that all ADTs bound in $\mathtt{st}$ be compatible with $st \uplus ext$. For scalars, on the other hand, we require that bindings in st be mirrored in $\mathtt{st}$, but we only require ext to be a submap of the remaining parts of $\mathtt{st}$. Intuitively, this means that we are allowed to forget about previously allocated scalar values, but that we must track ADTs precisely until we deallocate them.

3.2.3 Synthesis framework

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

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

This specification reads as follows:

- For all starting Facade states $\mathtt{st}$, if $\mathtt{st}$ is in relation with the Fiat state st extended by ext, then it is safe to run the Facade program $\mathtt{p}$ from state $\mathtt{st}$.

[9] In addition to a name string, named bindings are required to be accompanied by an instance of a type class mapping values of the type of bound computation in and out of the representation type used in Facade specifications.

- 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 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 1a), used to drive the proof search and provide the compilation architecture
- Connections between the $\rightsquigarrow$ relation and Fiat's semantics (for example the *bind* rule of figure 1b), used to reduce compilation of Fiat programs to that of Gallina programs
- Connections between Fiat and Facade, through the $\rightsquigarrow$ relation (such as the *if* rule of figure 1c). 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.

Furthermore, this relation has the pleasant property that it only manipulates Fiat states through the $\approx$ relation, under the same ext: this makes it automatically compatible with the equivalence relation derived from $\approx$ defined by connecting two Fiat states $st_1 \uplus ext$ and $st_2 \uplus ext$ when any Facade state is related to $st_1 \uplus ext$ iff it is related to $st_2 \uplus ext$. This makes the second category of lemmas easy to express, owing to the close parallel between Fiat computations and Fiat states, by reducing them to lemmas about this equivalence relation on telescopes.

3.2.4 Automatic extraction by synthesis

With these lemmas, the complexity of the compilation process is reduced to automating a proof-search problem by applying the right lemmas and introducing the right cuts.

Compilation proceeds in three steps: from a Fiat ADT specification, comprising multiple method written as Fiat computations mixing Gallina code with calls to Fiat ADTs, we produce a Facade module whose methods are yet undetermined; each of these methods is given a Hoare-style specification based on the $\rightsquigarrow$ predicate. For a Fiat computation f taking arguments $x_1 \ldots x_n$, this specification takes the following form:

$$\begin{array}{l}\exists\, \mathtt{p}.\ \forall\, x_1 \ldots x_n.\\ [\texttt{"x}_1\texttt{"} \leftarrow x_1] :: \\ \ldots :: \\ [\texttt{"x}_n\texttt{"} \leftarrow x_n]\end{array} \overset{\mathtt{p}}{\underset{\varnothing}{\rightsquigarrow}} [\texttt{"out"} \leftarrow \mathsf{f}\ x_1\ \ldots\ x_n]$$

From this starting point, extraction proceed by repeated pattern matching on the joint shapes of the pre and post-states. This pattern matching yields candidate compilation lemmas, which are 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 source drives the computation and of verification conditions, we do not have a program source to guide us; instead, we must rely on the shape of the pre- and postconditions. Just like the best programming languages prevent users from writing meaningless programs, the best specifications for such a synthesis task are those whose structure is regular enough to permit easy synthesis.

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 or by introducing an intermediate state after splitting the unknown program into two consecutive subprograms). 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 synthesised 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.

The compiler architecture is extensible: the main extraction 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 the following section[10].

3.3 Implementation details

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

The trusted base of this paper comprises three main parts: the Coq Kernel, a number of facts about ensembles represented as functions from elements to propositions (the derivations rely on ensemble extensionality, and use a number of admitted facts about combinations of ensembles), and a translation function between Facade specifications and telescope-based specifications (proving an equivalence between these two types of specifications is feasible on a case-by-case basis).

[10] Of course, since our focus is not on writing a general extraction mechanism (but instead on describing a convenient technique for extracting DSLs embedded in Gallina) these rules do not cover all possible Gallina programs. We are not aware, however, of patterns that would be fundamentally incompatible with this extraction strategy.

4. Evaluation

4.1 Micro-benchmarks

We started out by extracting a number of small Gallina programs performing tests and computing expressions using machine words, all of which are handled by the default library of compilation rules that comes with our extraction framework. In all these cases, the extraction process completes in a matter of seconds.[11]

We then extended the compiler by providing a trivial specification for a **`rand`** function (allowing it to return any number), proving a lemma that connected the Facade specifications of that function (expressed in unconstrained Hoare logic) with telescope-based specifications (expressed in terms of the $\rightsquigarrow$ relation). By adding a compilation rule to the extraction engine, in the form of one Ltac function, we were then able to extract the example given at the very beginning of this paper. The effort involved beyond the writing of the Fiat and Facade specification of the function was minimal, as the structure of the $\rightsquigarrow$ relation used to connect these two specifications makes it amenable to proof automation. In total, extending the compiler to support this external function whose behavior could not be obtained in pure Gallina required 20 lines of specifications, 5 lines of proofs, and about 10 lines to write the compiler extension.

Another convenient feature of optimizing compilers is intrinsics: although some constructs may be expressible in the source language, it may be interesting to give them special treatment while compiling. To replicate this pattern in our extraction engine, we added a rule to the compiler which, when spotting an application of a Gallina function, looks at the externally available functions to find one that matches the Gallina function[12]. Should such a function be found, compilation does not proceed by unfolding the definition of the function and examining its body, but instead emits a single function call. We made use of this construct to extract snippets doing bit-level manipulations on machine words – these operations would otherwise be hard to implement efficiently in Facade.

As another more advanced example of intrinsics, we added support to our extraction engine for maps and left folds[13], implemented as destructive while loops[14]. The rule for *foldl* is presented in Figure 2. Interestingly, while our extraction strategy has a strong flavor of synthesis, proving the lemmas

$$\frac{\forall h\ \mathsf{a}\ \ell.\ [\texttt{head} \leftarrow \textbf{ret}\ h] :: [\texttt{out} \leftarrow \mathsf{a}] :: t \overset{p_{body}}{\underset{[\texttt{ls} \leftarrow \ell] :: ext}{\rightsquigarrow}} [\texttt{out} \leftarrow \mathsf{f}\ \mathsf{a}\ h] :: t}{\begin{array}{l} p_{init}\texttt{;} \\ \texttt{end := empty?(ls)} \\ \textbf{While}\ \texttt{(not end)} \\ \quad \texttt{hd := pop!(ls)} \\ \quad p_{body} \\ \quad \texttt{end := empty?(ls)} \\ \texttt{delete!(ls)} \end{array}} \textsc{FoldL}$$

$$[\texttt{ls} \leftarrow \textbf{ret}\ \ell] :: t \overset{}{\underset{ext}{\rightsquigarrow}} [\texttt{out} \leftarrow \mathit{foldl}\ \mathsf{f}\ \mathsf{a}_0\ \ell] :: t$$

Figure 2: The *foldl* rule, reducing ℓ with f on initial value a_0

```
threshold ← rand();
ret (foldl (fun acc w ⟹
            if threshold ≺ w then acc
            else w :: acc)
      seq []))
```

```
random := std.rand()
out    := list[W].nil()
test   := list[W].empty?(seq)
While (test = 0)
  head  := list[W].pop(seq)
  test0 := random < head
  If Const 1 = test0 Then
    pass
  Else
    call list[W].push(out, head)
  EndIf
  test := list[W].empty?(seq)
call list[W].delete(seq))
```

Figure 3: Example of a micro-benchmark: the first part of the figure shows a Fiat program filtering out elements of a list falling below a randomly selected threshold; the bottom shows the extracted program. The derivation guarantees that when given a sequence of words *seq* as an argument `"seq"`, the resulting Facade program will end in a state containing a list of word permissible by the original Fiat specification, stored in the variable `"seq"`.

required to add support for high-level constructs such as maps and folds has a strong flavor of traditional imperative program verification, where Fiat telescopes serve as a specification language for Facade programs. This makes it relatively easy to prove these lemmas, as they are mostly series of deductions following Facade's semantics. This allowed us to compile examples such as filtering lists, or flattening lists of lists (compiling to nested while loops).

In total, our micro-benchmark suite contains about 25 programs manipulating words, lists of machine words, and lists of lists of machine words, with optional calls to a `random`

[11] The benchmarks can be run interactively by stepping through the `src/CertifiedExtraction/Benchmarks/Microbenchmarks.v` file in the attached source

[12] This matching is expressed by using parametrized specifications specialized to particular Gallina functions, connected by a single generic function

[13] This pattern of adding support for new language constructs illustrates that the source language of our extraction engine is a fluid DSL, including small parts of Gallina, and encompassing many programs beyond pure Gallina (the sources that we extract are nondeterministic Fiat computations).

[14] The destructive nature of loops is not an issue; it is easy to detect cases where a list is needed multiple times and to create a copy of it before iterating over it.

```
Definition SchedulerSpec : ADT _ :=
 QueryADTRep SchedulerSchema {
  Def Constructor0 "Init" : rep := empty,

  Def Method2 "Spawn" (r : rep)
      (new_pid cpu : W) : rep * bool :=
   Insert (<"pid" :: new_pid,
           "state" :: SLEEPING,
           "cpu" :: cpu>)
    into r!"Processes",

  Def Method1 "Enumerate" (r : rep)
      (state : State) : rep * list W :=
   procs ← For (p in r!"Processes")
           Where (p!"state" = state)
           Return (p!"pid");
   ret (r, procs),

  Def Method1 "GetCPUTime" (r : rep)
      (id : W) : rep * list W :=
    proc ← For (p in r!"Processes")
           Where (p!"pid" = id)
           Return (p!"cpu");
   ret (r, proc)
}.
```

Figure 4: The original Fiat specification of the 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. Fields in the final program are indexed by machine words, instead of the original strings.

method specified to return an arbitrary machine word. Examples of micro-benchmarks range from simple operations such as performing basic arithmetic, allocating data structures, calling compiler intrinsics, or sampling random numbers to more complex operations involving sequence manipulations, such as reversing, filtering (e.g. removing elements greater than a threshold), reducing (e.g. reading in a number written as a list of digits in a given base), flattening, duplicating or replacing elements (e.g. ensuring that a sequence contains no zeroes). A detailed example of such a micro-benchmark, and the corresponding code, is given in Figure 3.

4.2 Macro-benchmarks

To evaluate the performance and capabilities of our extraction engine on real-life scenarios, we targeted the query-structure ADT library of the Fiat paper [4], as well as an additional ADT modeling process scheduling (see Figure 4). The source programs in this collection of tests were full ADTs, collections of Fiat methods obtained by refining Fiat specifications expressed in an SQL-like language into computations intermixing Gallina terms and non-deterministic method calls to bag ADTs.

In all cases, the final result of extraction is a Facade module, a collection of exported methods, dependency declarations, and proofs. Each exported method corresponds to one of the methods of the original Fiat ADT, and is paired with an axiomatic specification inherited from the Fiat method and a proof of conformance to that specification[15].

Each dependency declared by the Facade module is expressed through an axiomatic specification, and resolved upon linking with a suitable implementation. Part of the extraction process consists in synthesizing Facade calls to the relevant external methods to implement calls made at the Fiat level to structures obeying the Fiat concept of a bag. These synthesis steps are justified by proving lemmas connecting the semantics each Fiat-level bag operations to axiomatically-specified operations on nested trees: thus, following a process similar to the one used in the intrinsics part of our extensibility micro-benchmarks, we wrote descriptions of the operations provided by Fiat bags in Facade terms, and proved lemmas connecting the two specs through the $\rightsquigarrow$ relation.[16]. Due to the structure of the source Fiat programs, we were able to use Facade specifications expressing nested tree operations in terms of mutations[17]. Finally, we added corresponding rules to the extraction engine.

The programs that we started from were similar to the following:

```
(* Find all processes in a given state
   and return their process IDs *)
a ← CallBagMethod BagFind table (_, STATE, _);
 ret (fst a, revmap (fun x ⟹ GetAttribute x 0) (snd a))

(* Find a process by process ID
   and return its running time *)
a ← CallBagMethod BagFind table (ID, _, _);
 ret (fst a, revmap (fun x ⟹ GetAttribute x 2) (snd a))

(* Insert a new process with id ID;
   fail if the key already exists *)
a ← CallBagMethod BagFind r (ID, _, _);
if length (snd a) = 0 then
 u ← CallBagMethod BagInsert (fst a) [d, SLEEPING, d0]
 ret (fst u, true)
else
 ret (fst a, false)
```

And the resulting, extracted programs were similar to the following (the other methods have been omitted):

```
snd := Tree.findByFirstIndex(table, d);
ret := List[W].new();
test := List[Tuple].empty(snd);
```

[15] This proof of conformance is lifted from the trace of the extraction process, which yields a witness expressed in terms of the $\rightsquigarrow$ relation

[16] These lemmas are generally straightforward, and mostly serve to bridge small encoding gaps (introducing the concept of mutations, for example) and to lift axiomatic specifications into operational specifications.

[17] Thus Facade's linearity requirement, which simplifies many aspects of extraction, did not prove limiting

```
While (Var test = Const 0)
    head := List[Tuple].pop(snd);
    pos := Const 2;
    head' := Tuple.get(head, pos);
    size := Const 3;
    call Tuple.delete(head, size);
    call List[W].push(ret, head');
    test := List[Tuple].empty(snd);
test := List[Tuple].delete(snd);
```

Interestingly, and in conformance with the Fiat idea 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: indeed, revmap can be implemented in a single pass over its input list in constant stack space, while map cannot.

After extracting the adts (extraction takes less than a minute in all cases), we compiled the resulting programs using the new Facade to Bedrock compiler. At that point in the process, we had obtained assembly programs with proofs of partial correctness with regard to the original SQL-flavored specifications, assuming availability of a number of external methods on nested treed.

All that remained to be done was to implement and verify these external dependencies. Facade, Cito, and Bedrock were all potential candidates; we chose Bedrock for performance, and used the Bedrock linker to obtain executable programs.

This table shows the running time in seconds for each of several tests of the extracted assembly code. For each value i in the #Pids column, we create i random processes with IDs from 0 to #Pids-1, giving each a random CPU ID in the same range. Then we measure the time for 2000 random Enumerate operations, which should be fast, as they are perfectly suited to the binary-search-tree structure we are using; and 1 random GetCPUTime operation, which can be quite slow, as it needs to visit every node of the outer search tree, calling a method on the inner search tree in each case.

#Pids	2000 Enumerates	1 GetCPUTime
10	.01	.00
100	.02	.00
1000	.02	.01
10000	.04	.88

We cannot resist pointing out that the benchmarks worked correctly on the first try, with no debugging time required for the verified code, only in our benchmarking harness.

5. Related work

Quite closely related to our work is a project by Lammich that uses Isabelle/HOL to do automated refinement of functional programs to an embedded imperative language called Imperative/HOL, which includes garbage collection (unlike our Facade) but exposes mutable objects. Our project and his proceeded independently at the same time, and there are both important similarities and differences between our solutions. Lammich's tool has been applied to derive implementations of classic textbook algorithms and other challenging cases, whereas our focus is on fully automatic derivation from SQL-style specs, and so we have only looked at starting specs that are simpler in some sense, along the lines of what real-world programmers write regularly in mundane applications. Both approaches use some kind of linearity checking to bridge the gap between functional code, where sharing of data structures is natural, and imperative code, where object identity matters. Lammich's approach reasons with separation logic [16] and axiomatic semantics, while we apply the lighter-weight approach of Facade's decidable syntactic checks on linear usage of variables, as a post-phase after we derive programs in a novel sequent-calculus style supporting nondeterminism, without explicit pointer reasoning. Furthermore, Lammich's tool only supports generating verified Imperative/HOL code, whereas our pipeline allows integration of synthesized imperative programs with data structures implemented in assembly language, compiled with proofs from any of the several languages in the Bedrock ecosystem, or any other for which someone writes a certified or certifying compiler. An important part of the larger context is that our translation has been successfully integrated into an automated, proof-generating pipeline from relational specifications to executable assembly code, and, as far as we know, no such pipeline has been presented before.

Another closely related project is by Myreen and Owens, on extracting terms written in the pure functional programming subset of the logic of HOL4 into programs written in a dialect of ML called CakeML. The main differences between that project and ours are in the language choices, external linking abilities, and optimization opportunities, in addition to a significant difference in focus. Indeed, the target language for the extraction procedure described by Myreen and Owens has semantics that are relatively close to that of the source language: both the pure functional subset of HOL4 and miniML are functional languages with relatively expressive type systems, and neither exposes memory management to the user. In contrast, our source language is more restricted, but also further remote from the language that we are targeting: this gives us many more opportunities for optimizations, including those related to memory management. We expose these opportunities to users of our compiler by letting them inject their domain-specific optimization knowledge into the compiler by proving compilation lemmas: in that sense, our work may be better thought of as a compilation library, providing a large degree of extensibility and flexibility to allow for compilation of domain-specific languages of various natures. Our approach further differs from that of Myreen and Owens by allowing us to compile partially refined Fiat programs, including dependencies on externally implemented method calls.

Beyond that, our project draws inspiration from a number of related efforts that can broadly be categorized into four groups: program extraction from proof assistants, compiler

verification, extensible compilation, and formal decompilation in proof assistants.

Program extraction Many recent verified software development efforts use program extraction to produce executable binaries from their code: examples include the verified C compiler CompCert [10], as well as the Ynot framework [14] for verifying Haskell-style monadic Gallina programs. Such extraction facilities are found in Coq [21] and other proof assistants, including Nuprl [2] and more recently F* [19], a proof assistant developed at Microsoft Research. These extractors are rather complex programs, subject to varying degrees of scrutiny: for example, the theory on which Coq's extraction is based was mechanically formalized and verified [11], but the corresponding concrete implementation itself was not subjected to verification. Our new compilation strategy does not suffer from these limitations: in particular, we ensure that the guarantees provided at the Gallina level are fully preserved as we move down to Facade, and from there down to Bedrock. As such, our method can be viewed as a sound and flexible alternative to extraction, which shines when the source language is small and presents patterns amenable to specific optimizations.

Compiler verification In addition to preserving semantics, our compilation strategy allows Fiat programs to take advantage of the external linking capabilities offered by Bedrock through Facade. This contrasts with work on verified compilers such as CompCert [10] or CakeML [7]: in the former, correctness guarantees indeed only extend to linking modules compiled with the same version of the compiler: crucially, CompCert does not allow users to link their programs against manually implemented and verified performance-critical software libraries. More 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 in the context of verified operating-system kernel development [17], is to use translation validation instead of compiler verification: much like our extraction mechanism proves a specific theorem for each derivation, these efforts focus on validating individual compiler outputs. This approach is particularly attractive in the case of existing compilers, but generally falls short when trying to verify complex optimizations. One of the most fascinating uses of translation verification is the work on seL4 [6], a microkernel verified using Isabelle/HOL. Security properties of seL4 are established by directly reasoning about deeply embedded C code, and these formal guarantees are propagated to assembly by using a mix of automated checkers and verification tools like SONOLAR [15] and Z3 [3].

Extensible compilation Multiple research projects have focused on providing optimization opportunities to users beyond the core of an existing compiler. Some of these projects, with the recent example of Racket's extensibility API [22], do not focus on verification. Other efforts, such as the Rhodium system [9], let users express and verify transformations in a domain-specific language for optimizations. Unfortunately, most of these tools are not themselves proven sound and have not been integrated in larger systems to provide end-to-end guarantees. One recent and impressive exception is the XCert project [20], which extends CompCert with an interpreter for an embedded DSL describing optimizations, allowing users to describe program transformations in a sound way. Parts of our approach can be related to this project by noting that we build our architecture on a collection of sound rewriting rules; in a sense, our entire compiler is written as a collection of transformations, each of which gets us closer to the final desired Facade program.

Formal decompilation A number of projects have used HOL-family proof assistants for automatic proof-generating translation of low-level to high-level code, which is something of an inverse to the kind of derivation that concerns us in this paper. Myreen et al. have carried out a significant line of work on decompilation from assembly code to HOL functional programs, where those programs are clean enough to reason about directly in further proofs. Greenaway et al. have applied a similar strategy to decompile C code into HOL functional programs. Decompilation is attractive for cases where the low-level code is fixed by external concerns or where it may be infeasible to verify the compiler used to produce that code, while the reverse direction of program derivation from specifications may be a better fit for from-scratch projects that invest in compiler verification.

6. 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 the sole kernel of the proof assistant. Beyond describing our technique and detailing its implementation, we have shown it to be suitable to the extraction of DSLs embedded in proof assistants: we first applied it on a series of micro-benchmarks and then used it to push the guarantees offered by nondeterministic programs refined by Fiat all the way down to the verification-aware assembly language Bedrock, *via* a new language designed to facilitate reasoning about memory allocation as we synthesized extracted programs. In the process, we have closed the last gap in the first mechanically certified translation pipeline from declarative specifications to assembly-language libraries, supporting user-guided optimizations and abstraction over ADT implementations.

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] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[3] L. de Moura and N. Bjrner. Z3: An efficient smt solver. In *Proc. TACAS*, pages 337–340, 2008.

[4] 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.

[5] 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.

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. SOSP*, pages 207–220. ACM, 2009.

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

[8] 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. URL http://dx.doi.org/10.1007/978-3-319-22102-1_17.

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

[10] 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.

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

[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] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *Proc. ICFP*, pages 229–240. ACM, 2008.

[15] J. Peleska, E. Vorobev, and F. Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 298–312. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-20397-8. doi: 10.1007/978-3-642-20398-5_22.

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

[17] 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.

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

[19] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, and J.-K. Zinzindohoue. Dependent types and multi-monadic effects in F*. Draft, July 2015.

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

[21] The Coq Development Team. The Coq proof assistant reference manual, version 8.4. 2012.

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

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

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