Compilation using Correct-by-Construction Program Synthesis

by

Clément Pit-Claudel

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

Author ....
Department of Electrical Engineering and Computer Science
August 30, 2016

Certified by ....
Adam Chlipala
Associate Professor without Tenure
Thesis Supervisor

Accepted by ....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Compilation Using Correct-by-Construction Program Synthesis

by

Clément Pit-Claudel

Submitted to the Department of Electrical Engineering and Computer Science
on August 30, 2016, in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

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 explore a novel approach to extracting and compiling embedded domain-specific languages developed in a proof assistant (Coq), showing how it allows us to extend correctness guarantees all the way down 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 of externally implemented data structures, delaying linking to the assembly stage. Composed with the Fiat and Bedrock frameworks, our new method provides the first proof-generating automatic translation from SQL-style relational programs into executable assembly code.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor without Tenure

Contents

List of Figures

Chapter 1

Introduction

Program errors, or bugs, are a well-known and costly reality of software development. Bugs arise from a wide variety of situations, such as simple programming mistakes, incorrect assumptions about program state, improper handling of exceptional situations, misconfigurations, incorrect assumptions about other software components, miscompilations, etc.; accordingly, significant research and engineering effort has been dedicated to devising techniques to reduce software errors caused by incorrect programming, yielding tools and concepts such as design patterns, software abstraction, encapsulation, compile- and run-time checks, type systems, program verification, and program synthesis. Languages that strive to reduce the possibility of programming errors by exposing programmers to abstractions remote from a machine's actual workings are usually described as *high-level languages*; conversely, languages that mostly provide access to primitive machine operations are usually called *low-level*. High-level languages offer significant productivity benefits and eliminate large classes of errors (incorrect memory accesses, certain types of memory leaks, certain classes of data races, etc.).

These benefits, unfortunately, most often come with significant costs (garbage collection, array bounds checks, dynamic type checks, coarse-grained locking, etc.). Even when languages offer so-called *zero-cost abstractions* (high-level patterns guaranteed to have no additional run-time costs over their low-level counterparts), they still put constraints on the programs that users may express: by abstracting away some of the machine's bare-metal characteristics, they make certain sequences of machine operations inexpressible or expressible at higher cost (two famous examples of such abstraction-breaking patterns are presented in figure 1.1). Some languages work around this by offering unsafe blocks or *foreign-function* interfaces to lower-level languages (C# has `unsafe` sections, OCaml and Haskell have arbitrary casting typing operators `Obj.magic` and `unsafeCoerce`, C++ allows `reinterpret_cast`-ing, and C lets programmers use inline assembly); in both cases, however, the guarantees of the high-level language do not extend to the low-level code fragment, and the boundary introduces another opportunity for subtle errors.

Complex, low-level code in real-life systems is often the result of multiple rewrites, reimplementing performance-critical sections in lower-level languages and hand-tuning the resulting programs. Given this, it is an easy step to imagine that such complex code could be derived by applying a series of transformations: manually (programming by refinement), or even better automatically (as is to some extent done by multiple-passes compilers internally and by various other types of interpreters and code generators: profile-guided optimizers, auto-tuners, superoptimizers, etc.).

But the general-purpose automatic code generation found in mainstream compilers often falls short, in terms of performance, of human-tuned low-level programs: programmers frequently have highly domain-specific (or even program-specific) insight that the compiler does not have access to. Various mechanisms have thus been proposed to let users leverage such specific insight: domain-specific languages with more targeted optimizations and better configurability (SQL and database indices, for example), embedded domain-specific languages to handle well-delimited tasks inside of larger programs (regular expressions, Microsoft's language-integrated queries), or compiler-provided hooks to

```c
void send (short *to, short *from,
           int count)
{
  int n = (count + 7) / 8;
  switch (count % 8){
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
               } while (--n > 0);
  }
}
```

```c
float Q_rsqrt(float number)
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;

  i  = *(long*)&y;
  i  = 0x5f3759df - (i >> 1);
  y  = *(float*)&i;
  y  = y * (threehalfs - (x2 * y * y));

  return y;
}
```

Figure 1.1: Example of low-level programs whose logic is harder to express in high-level languages. Left: Duff's device (valid in C and C++) implements an unrolled loop with no separate setup code by transferring control directly into the body of the loop, at an offset based on the remainder modulo 8 of its argument. Right: Quake 3's fast inverse square root computes a close approximation of $^1/_{\sqrt{x}}$ by doing integer arithmetic on the bits of an IEEE floating point number.

introduce new optimizations at various levels (often in the form of same-language optimization passes). All these methods, however, suffer from a common problem: extensibility. Each mechanism can be thought of as a domain-specific language of optimizations, and each of these languages makes expressing certain optimizations easy but leaves others hard (if not impossible) to express. A parser-generation library, for example, might not support the right parsing algorithm; a regular-expression library may have exponential complexity in corner cases; an SQL engine might miss an opportunity to use an index; and a particular optimization might not be expressible in a compiler's rewriting language or may not fit in one of the already-existing hooks. Fundamentally, one of the core issues is that these tools are separated into two components: core infrastructure and user-programmable extensions. Even in the best-designed cases, compilers are not fully open to user modification. The result is disappointing: program performance can be improved up to a point, but critical sections need to be rewritten in a low-level language when the limits of tunability are reached.

Such an architecture makes sense, of course. The internals of a compiler, as well as existing extensions, might rely on subtle invariants; allowing users to inspect and modify these internals in arbitrary ways runs the risk of letting them introduce subtle and complex errors. Verification, fortunately, offers a different route.

Progress in tactic languages, a way to program proof assistants to discover complex proofs or drive complex proof-producing processes, has made it practical to build tools that leverage general-purpose proof assistants to synthesize correct-by-construction programs: concretely, automated reasoning is used to derive programs with little to no user intervention, while guaranteeing that the final results respect certain constraints. Fiat (Delaware et al. 2015), for example, leverages the programming-by-refinement methodology to offer an extensible framework for building domain-specific languages and proof-producing compilers for these languages. It has much of the flavor of manual programming by refinement, thus allowing users to leverage arbitrarily complex optimizations to transform high-level programs into efficient functional code, but it also introduces lightweight automation to take away most of the pain of the process: a query planner in Fiat, for example, would not be a monolithic engine with predetermined extension points; instead, it would be a collection of refinement procedures (each guaranteeing correctness of individual transformations by producing machine-checkable proofs), assembled into a compiler. To some extent, compiler designers and users are placed on equal footing: users can verify arbitrarily complex optimizations and insert them at arbitrary points in the compilation process.

But Fiat solves only part of the problem. Indeed, Fiat is fundamentally a single-language tool: all Fiat programs are encoded as Gallina terms (Gallina is the programming language of the Coq proof assistant), making it hard to express low-level optimizations in Fiat and weakening Fiat's correctness guarantees (Fiat programs are extracted into OCaml and compiled to machine code using unverified software components). There are good reasons for these two limitations: first, embedding itself in Gallina lets Fiat inherit its semantics and thereby leverage Coq's reduction machinery and powerful collection of primitive tactics (primitive tactics, such as `rewrite` and `simpl`, are programs implementing small steps of mathematical reasoning). Second, certified extraction is a complex and, unlike certified compilation, only recently studied problem: the fundamental issue can be summarized by saying that a proof assistant's logic cannot directly be used to reason about itself. Since Fiat programs are written in Gallina, a compiler from Fiat to another language cannot be written in Gallina.

This thesis overcomes these difficulties by introducing a novel proof-producing compilation technique, deriving imperative implementations of functional Fiat programs by recasting the compilation task as a synthesis problem and using lightweight Hoare-style pre- and postconditions to drive the proof-search process. This novel approach is extensible and safe: users can express arbitrary *cross-language* optimizations, with no risk of jeopardizing the correctness of the final programs. It bypasses the issue of certified extraction by producing machine-checkable correctness certificates with each compiled program: these certificates are type-theoretic proofs, whose elaboration is done as the program is being constructed. Furthermore, our approach makes it straightforward to build domain-specific compilers from collections of verified translation components, each tasked with handling specific patterns appearing in source programs. By reassembling these pieces of logic, or writing new ones, our users tailor extraction to their own domains and programs, maximizing performance while ensuring correctness of the resulting low-level programs.

Verification is usually seen as a high-cost burden on otherwise rapid program-development cycles, but in our case verification is an asset: working in a proof assistant allows us to aggressively optimize input programs, while relying on a small trusted logical core to reject soundness-breaking transformations. Thus we never shun optimizations for fear that they might be incorrect: any optimization is fair game, as long as we can justify its correctness in the particular context of the specific program on which we wish to apply it.

This compilation technique allows us to bridge the last gap in the implementation of a long-standing vision: a complete stack transforming high-level specifications into efficient assembly programs, while sacrificing little to no performance and guaranteeing correctness (the flavor of correctness that we guarantee is *partial correctness*: we produce proofs that our compiled programs do the right thing under the assumption that they terminate, but we do not guarantee termination). Composed with the Fiat and Bedrock frameworks, our new method indeed provides the first fully proof-generating automatic translation from SQL-style relational programs into executable assembly code, bottoming out in fast and lightweight assembly programs.

1.1 Components

To carry out this vision, we segment the translation from specifications to assembly into a sequence of smaller steps. At the top lies Fiat (Delaware et al. 2015), a framework for writing domain-specific languages in the Coq proof assistant (The Coq development team 2015). Specifications written in these domain-specific languages are translated and optimized by refinement to produce efficient, potentially nondeterministic, purely functional programs. At the bottom of the stack lies Facade, an Algol-like imperative language that ensures linear use of heap-allocated data and can thus expose memory as an unordered collection of names and values. Facade compiles to Cito (a very similar language in which the linearity constraint is relaxed to allow pointer aliasing), and Cito compiles to Bedrock, a verification-aware assembly language that can straightforwardly be extracted to x86 assembly (this simple step is currently unverified). All of these parts are previous work: in this thesis, we describe the design and implementation of a proof-producing compiler between refined Fiat programs and Facade. The following paragraphs give a more detailed overview of each part of our stack.

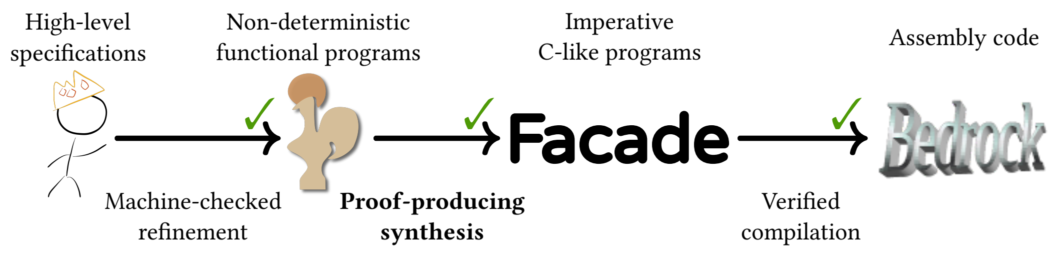
High-level specifications
Non-deterministic functional programs
Imperative C-like programs
Assembly code
Facade
Bedrock
Machine-checked refinement
Proof-producing synthesis
Verified compilation

Figure 1.2: Overview of the Fiat-to-Assembly pipeline

1.1.1 Top: Fiat

At its core, Fiat is simply a collection of Coq tactics and lemmas to reason about the nondeterminism monad (Figure 2.1). Authors of Fiat programs mix purely functional Gallina code with values obtained through Fiat's `Pick` operation (which models the selection of a value satisfying an arbitrary predicate) to produce shallowly embedded programs and specifications. For example, a Fiat programmer might write a specification of the following form:

```
Definition approximate_fixpoint (f: ℝ → ℝ) (ε: ℝ) : Comp (ℝ * ℝ) ≜
  (** Compute an ε-approximate fixpoint ‘x’ of ‘f’
      and return the pair (x, f x) **)
  x ← {x | abs (f x - x) ≤ ε};
  ret (x, f x)
```

Library authors using Fiat are then free to define domain-specific combinators that desugar to uses of `Pick` and *bind*, while exposing a nicer syntax for a given domain. For example, one might define the following simple DSL:

```
Notation FIXPOINT f ≜ {x | f x = x}
Notation ARGMAX   f ≜ {x | ∀ y, f y ≤ f x}
Notation ARGMIN   f ≜ {x | ∀ y, f y ≥ f x}
```

Concretely, Fiat has been used to define DSLs for SQL-style queries, BNF-style grammars, and binary encoders (serialization routines); work is planned on using it to derive SMT-style checkers.

High level specifications are then refined, using domain-specific automation written in Ltac (Coq's tactic language), to produce efficient functional programs. This refinement step substitutes most uses of `Pick` with programs whose outputs satisfy the `Pick` condition (for example, one might refine `{l' | sorted l' ∧ Permutation l l'}` with `ret (mergesort l)`).

In most cases, the resulting programs are thus directly executable, deterministic Gallina programs, which can be extracted and executed using Coq's extraction mechanism and the standard OCaml compiler—this is the approach followed by the original Fiat paper (Delaware et al. 2015).

In more complex cases, however, it can also be useful to leave some `Pick`s in the final program, thereby leaving responsibility for full refinement to later stages of the pipeline (this is discussed in more detail in the next chapter).

1.1.2 Bottom: Bedrock, Cito, Facade

Bedrock is a fairly typical assembly language, extended with precise semantics that allow programmers to reason about assembly programs. Bedrock functions are specified using Hoare-style pre- and postconditions expressed in separation logic, and a linking theorem allows programmers to combine individual Bedrock functions into larger units. An example Bedrock program taken from Bedrock's current sources and described in a 2013 paper (Chlipala 2013), together with a specification and proof of correctness, is given below:

```
Definition swapS : spec ≜ SPEC("x", "y") reserving 2
  Al v, Al w,
  PRE [V]  V "x" ↪ v  ∗  V "y" ↪ w
  POST[_]  V "x" ↪ w  ∗  V "y" ↪ v.

Definition swap ≜ bmodule "swap" {{
  bfunction "swap" ("x", "y", "v", "w") [swapS]
    "v" ←* "x";
    "w" ←* "y";
    "x" *← "w";
    "y" *← "v";
    Return 0
  end
}}.

Theorem swapOk : moduleOk swap.
  vcgen; abstract sep_auto.
Qed.
```

Cito is a much-higher-level imperative language by Wang et al. (2014), featuring relatively straightforward imperative semantics and offering a compiler to Bedrock with a machine-checked proof of partial correctness. Cito's main strengths are its support for separate compilation and its handling of function calls: a Cito program can make either operational calls, which transfer control to a Cito method whose body is explicitly known, or axiomatic calls, which transfer control to a method described only in terms of input-output specifications. Such axiomatic calls are resolved at link time, by providing for each such call a concrete implementation verified to respect the input-output specifications that the source program relies on. The following code samples present an example of such an axiomatic specification for a logical left-shift operation, as well as a Cito program making an axiomatic call to this specification, assuming that the name `Intrinsics.LSL` points to it.

```
Definition lsl : AxiomaticSpec unit.
  refine {|
      PreCond  ≜ λ args ⇒
        ∃ x, args = [SCA _ x];
      PostCond ≜ λ args ret ⇒
        ∃ x, args = [(SCA _ x, None)] ∧
             ret = SCA _ ($2 ⊗ x)
    |}; crush_types.
Defined.
```

```
output ≜ DCall Intrinsics.LSL(input)
```

Facade, finally, shares almost all of Cito's syntax but exposes very different operational semantics. Using a collection of syntactic checks on input programs, Facade ensures that each heap-allocated structure is only pointed to by a single name and that no heap-allocated memory is leaked. This allows Facade to present to the user a simple interface to memory, as a collection of names and values. An inductive proof guarantees that injecting Facade's syntax into Cito yields a valid Cito program with equivalent partial-correctness semantics. An example Facade program is presented below:

```
offset ≜ 0
len ≜ List[W].length(lst)
max ≜ List[W].get(lst, offset)
While (offset < len)
  elem ≜ List[W].get(lst, offset)
  If elem > max Then
    max ≜ elem
  Else
    __
  EndIf
  offset ≜ offset + 1
EndWhile
```

1.1.3 Middle: Functional to imperative compiler

The missing piece of this stack is a Fiat-to-Facade compiler: a procedure capable of producing, given a Fiat program, an equivalent Facade program[1]. This translation should be correct and extensible: users should be able to inject their own domain-specific insight and suggest problem-specific optimizations.

1.2 Going from Fiat to Facade

This Fiat-to-Facade translation task is atypical and presents a number of challenges, discussed below:

[1]This equivalence is the subject of a precise definition given in section 4.1.6; informally, it guarantees that the Facade program (if it terminates) outputs a value whose machine representation is the low-level equivalent of one of the results allowed by the original high-level specification, according to a user-supplied equivalence. The next section introduces the task and challenges in more detail.

1.2.1 Challenges

This is not a traditional verified compiler

Typical verified compilers consume and produce deeply embedded programs. That is, verified compilers are semantics-preserving transformation mapping each abstract syntax tree (AST) of a source language (encoded as a value of an inductive type) into an AST of a target language. Our task, on the other hand, is often described as extraction, or reflection: it is, to some extent, the reverse of denotation: we want to produce a procedure that takes as input a shallowly embedded Fiat program and produces a deeply embedded Facade program as the result.

Such a program cannot be expressed in Coq's Calculus of Inductive Constructions (CIC): letting a Gallina program inspect the syntax used to construct its arguments (a process usually called "reflection") would be soundness-breaking, as it would break the various conversion rules of the CIC. In particular, it is true in Coq that if a (possibly open) term a reduces to a', then for any function f, $(f\ a)$ reduces to $(f\ a')$. Thus a Gallina function must return the same value regardless of how reduced its input is. Concretely, this means that a Gallina program cannot discriminate between `let x ≜ f() in x + x` and `f() + f()`, making most forms of interesting reflection impossible[2].

There is no hope, therefore, to produce a Gallina function translating ("extracting") Fiat programs into Facade programs; instead, we need to manipulate Fiat programs externally, using an unverified procedure to produce our compiled Facade programs.

The compiler must be correct

Not being able to write the translation in Gallina, however, should not prevent us from ensuring that the resulting programs are correct: indeed, instead of proving the compiler

[2]Another way to understand this is to notice that the only discriminating construct in the CIC is `match`, and `match` discriminates on values (not on syntax)—and even then only on values of inductive types.

correct, we can certify each of its runs. Thus, we want the compiler to be *proof-producing* instead of *proven-correct.*

The final correctness statement that we wish to obtain is one that connects the output of the compiled Facade program to a subset of the values allowed by the (nondeterministic) source program.

Source programs may not be directly executable

The structure of refined Fiat programs prevents us from using Coq's built-in extraction: not only is this procedure excessively rigid, but it also does not handle our embedding of the nondeterminism monad. Indeed, due to the way in which Fiat programs are encoded (as functions from values to propositions), extraction erases Fiat terms entirely (Figure 1.3).

```
Definition PickDoubleOrNext (input: ℕ) : Comp ℕ ≜
  {x | List.In x [2 * input; 2 * input + 1]}.

Extraction PickDoubleOrNext.
(* type pickDoubleOrNext = __ *)
```

Figure 1.3: An example Fiat program that multiplies its input by two and optionally adds one. Extracting this program yields an empty type, instead of a program returning either of the permitted values (`Comp ℕ` is the same as $\mathbb{N} \to \mathbb{P}$, which the extraction engine erases).

To circumvent this issue, our compiler must recognize the semantics of Fiat's encoding and produce a Facade program whose results are permitted by the original programs.

Low-level representations are user-specified

To preserve abstraction, we do not want to constrain high-level specifications to be written in terms of low-level data structures: for example, we wish to allow users to formulate their high-level specifications in terms of purely functional data structures such as inductively defined linked lists, sets or multisets represented as functions from a type to `Prop` or $\mathbb{N}$, etc. Similarly, we want to allow users to reason about bounded natural numbers and Booleans naturally, instead of having to manipulate an explicit encoding into machine words.

On the other hand, to achieve optimal performance and interoperability, we do want the low-level programs to manipulate simple imperative structures, instead of defining an analogue of each inductive structure used in the high-level specifications or in the refined Fiat programs. Thus we need users to be able to specify injections from high-level data types into the types exposed by Bedrock through Facade, and we need to translate manipulations of the original, pure structures into operations offered by the low-level structures.

The compiler must be extensible

Traditional compilers are constructed as sequences of passes, alternating same-language rewriting with translation into lower-level languages. In this setting, extensibility is often achieved by providing hooks into the rewriting phases, allowing the most daring users to add new rewriting rules.[3] The translation phases, on the other hand, are seldom extensible (in a functional context, these phases are usually implemented via recursive descent over the source ASTs).

Limiting extensibility to rewritings is convenient but restrictive: on the one hand this decomposition helps ensure correctness (as long as each rewriting preserves the semantics of the program being transformed, adding rewritings cannot jeopardize the correctness of the final implementation), but on the other hand many optimizations cannot be captured

[3]GHC, for example, allows user programs to include special rewriting pragmas, such as `RULES "map/map" ∀ f g xs. map f (map g xs) = map (f ∘ g) xs`.

as rewritings without introducing intermediate languages.

We do not want to impose such a restriction on the compilation process translating Fiat to Facade. Instead, we wish to let users teach the compiler about new cross-language optimization opportunities (complex optimizations that are not simply representable as rewritings in either the source or target language), without staking correctness and without having to introduce an intermediate language mixing Facade's imperative semantics with Gallina's.

Additionally, we do not want to artificially constrain optimizations by requiring that the conditions under which they are applied be completely decidable predicates expressible in Gallina; instead, since we are already in the context of a proof-producing compiler, each optimization should be allowed to produce a proof of correctness tailored to the particular program being compiled and the particular instance of the optimization being applied.

As a concrete example, a useful optimization could be to translate a purely functional fold over a list of scalars into a destructive iteration over a mutable stack (as long as the original list is not reused later on in the program).

Facade doesn't have a garbage collector

Facade is an imperative language with manually managed memory. Though it exposes a simplified view of memory (as a map of names to values), it does not automatically handle deallocation of data structures after they fall out of scope. Instead, it requires programs to ensure that all newly allocated memory has been properly deallocated before returning control to the caller (with the exception of the return value, of course). This flexibility is convenient, but it introduces additional challenges: our compiler must precisely track allocated memory and ensure that the relevant destructors are called before returning.

1.2.2 A new twist on compiler design

Taken together, these challenges call for a new approach to compiler design: instead of implementing them as monolithic translation functions, we assemble domain-specific compilers from collections of small and specialized routines, each tasked with recognizing a particular pattern of the source language. The correctness of these routines need not be established, as they only drive the compilation process by contributing to an ongoing dialogue with the proof assistant: this ensures that mistakes in compilation routines cannot imperil the correctness of the final program.

This thesis describes a concrete, lightweight implementation of this new approach. We start by presenting some background and details (Section 2) about existing systems that we reuse. We then give an overview of our technique (Section 2), before describing it in full detail (Section 4), along with implementation notes (Section 5). We finish by presenting concrete results (Section 6) obtained using this technique, discussing alternative approaches and related work (Section 7), and concluding with directions for future work and final remarks.

Chapter 2

Background

Before presenting the details of our Fiat-to-Facade translation, we give further information about the source and target languages and highlight issues with traditional extraction.

2.1 Deductive synthesis in Fiat

Fiat programs are encoded using a type `Comp A` of computations returning values of type `A` and written using the usual `Bind` and `Return` monad combinators (Wadler 1995), as well as an additional nondeterministic choice operator `Pick` that lifts a function `A` $\to \mathbb{P}$ (which can be though of as describing a set of values of type `A`) into a computation (in Fiat's encoding, `Comp A` is exactly `A` $\to \mathbb{P}$, and `Pick` is simply the identity function). Figure 2.1 displays the definitions of Fiat's `Pick`, `Bind`, and `Return` combinators.

Fiat additionally defines a `computes_to` relation, to express that a value is permitted by a computation. A Fiat program `P` computes to a value `a`, denoted `P` $\leadsto$ `a`, iff the proposition `(P a)` holds (which is sometimes alternatively written as `a` $\in$ `P`, in line with the set interpretation assigned to `A` $\to \mathbb{P}$).

Finally, Fiat defines a notion of refinement: a computation C_2 refines a computation C_1

```
Definition Comp ≜ @Ensemble.
Definition Return : A → Comp A ≜ Singleton A.
Definition Pick (P : Ensemble A) : Comp A ≜ P.
Definition Bind (ca : Comp A) (k : A → Comp B) : Comp B ≜
  λ b ⇒ ∃ a, a ∈ ca ∧ (k a) ∈ b.
```

Figure 2.1: Fiat's encoding of the nondeterminism monad. `Ensemble A` is defined in Coq's standard library as `A → ℙ`.

if C_2 admits only a subset of the values permitted by C_1 (i.e. `∀ a,` $C_2 \rightsquigarrow a \to C_1 \rightsquigarrow a$, which could alternatively be written $C_2 \subseteq C_1$).

Starting from high-level specifications, such as SQL queries or parser grammars, Fiat programs are refined to suppress most nondeterministic choices and introduce more efficient operations. Additionally, Fiat programs, when part of a Fiat ADT (abstract data type, a collection of methods), may operate on internal state whose concrete type can be changed as long as this transformation does not affect the output of the ADT's methods (Delaware et al. 2015).

The tactics driving the refinement, often referred to collectively as an "optimization script," select an appropriate type for the ADT's internal state and take advantage of the available nondeterminism in the program being refined to make efficient implementation choices. For example, an optimization script might refine `{x | List.In x (map f sequence)}` into `ret (f (hd sequence))`, provided that it can prove that `sequence` is nonempty.

Removing all nondeterminism, however, is not always the best that a well-written optimization script can do. In some cases, nondeterministic choices can be left in the refined program to encode calls to axiomatically specified, externally implemented data structures and functions. As an example, consider plugging a verified key-value store implementation into a Fiat refinement. The Fiat program might reason about such a collection as an abstract function `K → option V`: the result of enumerating a store's keys could then be described as `{keys | ∀ k, List.In k keys ↔ ∃ v, store k = Some v}`.

- On the one hand, if the key-value store is implemented as a verified Gallina module exposing a `keys` function returning a list of the keys present in the store, then the type of maps could be changed from abstract functions to the module's storage type, and the nondeterministic choice above could then be refined into `ret (keys store)`. This would, however, prevent further stages of the compilation chain from swapping that Gallina implementation for a potentially more efficient implementation, unless that other implementation produced keys in the very same order as the verified Gallina implementation available at refinement time. Indeed, by choosing `(keys store)`, the refinement process would have committed to the specific ordering used by the Gallina implementation of `keys`.

- On the other hand, the refinement script could take advantage of the wiggle room left by the `Pick` to allow linking the refined programs, after compilation down to a lower-level language, against various data structures (even data structures that do not admit canonical representations — that is, data structures such as Union-Find that may admit multiple distinct internal states for a single high-level representation). Concretely, one would preserve the nondeterminism throughout the refinement and subsequent compilation process and leverage it to select, at link time, an implementation of a key-value store whose `keys` method does not guarantee a particular return order.

This last point is crucial. Fundamentally, a `Pick` gives us two types of nondeterminism: one that allows us to pick different deterministic implementations, and one that allows implementations to rely on private state to produce different answers when given what appears to be the same input at the Fiat level. Concretely, although the value `(remove k` `(add k v store))` may be indistinguishable at the Fiat level from the unmodified value `store`, an implementation might rely on internal, mutable data structures whose state is affected by the succession of calls to `add` and `remove`. This is a common pattern: in general terms, the order in which commutative or canceling operations are applied to a data structure may make no difference in terms of high-level representations while making a difference in terms of internal states of low-level implementations. Nondeterministic specifications allow these low-level implementations to safely expose their internal state and answer requests with optimal performance; it also makes it possible to link against implementations that do not admit canonical representations, such as implementations of disjoint-set data structures based on forests.

Figures 2.2, 2.3, and 2.4, authored by Adam Chlipala, present a full example of a simple Fiat derivation.

```
Require Import Tutorial.

(** A specification of what it means to choose a number
    that is not in a particular list *)
Definition notInList (ls : list ℕ) ≜
  {n : ℕ | ~In n ls}.

(** We can use a simple property to justify
    a decomposition of the original spec. *)
Theorem notInList_decompose ls :
  (upper ← {upper | ∀ n, In n ls → upper ≥ n};
   {beyond | beyond > upper})
  ⊆ (notInList ls)
Proof.
  refines; firstorder.
Qed.
```

Figure 2.2: Specifications for a simple Fiat derivation; the aim is to refine a program that, given a list of numbers, produces a number that does not belong to that list. Figure 2.3 shows relevant utility lemmas, and figure 2.4 shows the final derivation.

```
(** A simple traversal will find the maximum
    list element, which is a good upper bound. *)
Definition listMax ≜ fold_right max 0.

(** …and we can prove it! *)
Theorem listMax_upperBound init ls :
  ∀ n, In n ls → fold_right max init ls ≥ n.
Proof.
  induction ls; simpl; intuition; try arithmetic.
  apply IHls in H₀; arithmetic.
Qed.

(** Now we restate that result as a computation refinement. *)
Theorem listMax_refines ls :
  ret (listMax ls) ⊆ {upper | ∀ n, In n ls → upper ≥ n}
Proof.
  refines; apply listMax_upperBound.
Qed.

(** An easy way to find a number higher than another: add 1! *)
Theorem increment_refines n :
  ret (n + 1) ⊆ {higher | higher > n}.
Proof.
  refines; arithmetic.
Qed.
```

Figure 2.3: Utilities required to complete the derivation whose specifications are shown in figure 2.2.

```
(** Let's derive an efficient implementation.
    Intermediate goals are italicized and preceded by [⊢]. *)
Theorem implementation :
  { f : list ℕ → Comp ℕ | ∀ ls, (f ls) ⊆ (notInList ls) }.
Proof.
  begin.

  ⊢ (?f ls) ⊆ (notInList ls)

  rewrite notInList_decompose.

  ⊢ (?f ls) ⊆ (upper ← {upper : ℕ | ∀ n : ℕ, In n ls → upper ≥ n};
               {beyond : ℕ | beyond > upper})

  rewrite listMax_refines.

  ⊢ (?f ls) ⊆ (upper ← ret (listMax ls);
               {beyond : ℕ | beyond > upper})

  setoid_rewrite increment_refines.

  ⊢ (?f ls) ⊆ (upper ← ret (listMax ls);
               ret (upper + 1))

  monad_simpl.

  ⊢ (?f ls) ⊆ (ret (listMax ls + 1))

  finish honing.
Defined.

(** We can extract the program that we found
    as a standlone, executable Gallina term. *)
Definition impl ≜ Eval simpl in projT₁ implementation.
  = λ l : list ℕ ⇒ ret (listMax l + 1)

Eval compute in impl [1; 7; 8; 2; 13; 6].
  = ret 14
```

Figure 2.4: The final Fiat refinement, leveraging utility lemmas proven in figure 2.3.

2.2 Verified compilation to Bedrock

Verified compilation into low-level assembly is a well-studied research domain (Kumar et al. 2014; Leroy 2006); still, few projects support separate compilation smoothly (Gu et al. 2015; Stewart et al. 2015).

2.2.1 Semantics of Facade

Facade (Wang 2016), the language that our compiler targets, is a recent extension of Cito (Wang et al. 2014) that facilitates reasoning about memory allocation by exposing a memory as a map of keys to values.

Facade programs are packaged in *modules*, augmented with various proofs to form *compile units*; each compile unit comprises a collection of possibly mutually recursive methods, along with *exports* (axiomatic specifications of these methods for use by the module's clients), *imports* (specifications of all external functions that the module's methods depend on), and various proofs used internally by the compiler. These proofs guarantee many facts about each method of the module (packaged as `DFFun` entries in the `DFModule` record, which lives within the `CompileUnit` record). A subset of these facts is summarized below:

1. Method arguments are not overwritten by direct assignments (`ret_not_in_args`, `no_assign_to_args` in `DFacade.v`'s `OperationalSpec`).

2. Compiler-reserved names do not appear in method bodies, nor in method arguments or return variables (`args_name_ok`, `ret_name_ok`, `syntax_ok` in `Operational-Spec`).

3. Method arguments have unique names, no uninitialized variables are read, function calls have the right numbers of arguments, and the sum of the number of local

variables and the depth of the function body is less than 2^{32} (`is_good_func`).

4. The axiomatic specification of each method is refined by the operational semantic of its body (`ops_refines_axs`, whose actual definition is very close to that of `ProgOk` presented in section 4.1.6); this entails guaranteeing that the program is safe to run in initial conditions satisfying the method's axiomatic preconditions and that it then yields a return value permitted by the method's axiomatic postcondition. This also guarantees, as a side condition of safety, facts such as the following (`SafeCallOp`, `SafeCallAx`):

    - All functions that the method calls to are properly imported—that is, each axiomatic call to an external function should be to one of the functions listed in the module's imports.
    - Memory is never leaked by the input program: variables containing a reference to a block of heap-allocated memory are never on the left side of an assignment.

Most of these proofs (1, 2, 3) are reflective (Boutin 1997): each of them checks a decidable property of a Facade program or module, and the corresponding goals can thus be discharged by computation.

Reasoning about Facade programs

The semantics of Facade programs are described through two predicates, `Safe` and `RunsTo`, which characterize the conditions in which it is safe to run a program and the results of running it:

- `Safe` is a coinductive predicate taking a program and an initial memory state and asserting that running the program in that starting state will never cause it to get stuck (thus the program won't access undefined memory, nor call a method without ensuring that its preconditions are verified; it may, however, loop indefinitely).

- `RunsTo` is an inductive predicate taking a program, an initial state, and a final state; it describes the operational semantics of the Facade programming language.

These predicates are phrased in terms of Facade *states*, finite maps of strings to Facade *values*, themselves either scalars (32-bit stack-allocated machine words) or ADT values (in Facade parlance, an ADT value is an arbitrary heap-allocated object, constructed, manipulated, and deallocated through externally supplied functions). Bedrock and Facade developments are parametric in the choice of type used to represent heap-allocated values: the only constraint is that libraries being linked together agree on this choice of type. In general, one picks a sum type in which each alternative described a distinct low-level data structure. Here is an example of such a type:

```
Inductive ADTValue ≜
| WordList (ws : list W)
| WTupleList (ts : list (list W))
| ByteString (capacity: W) (bs: string)
```

Compiling Facade programs

Once a fully constructed module has been assembled, one can invoke the Facade compiler to produce a Cito module. Feeding this module to the Cito compiler produces low-level Bedrock assembly code, which can then be soundly linked against verified implementations of any external methods that the code depends on, yielding a closed Bedrock library guaranteed to obey the very same axiomatic semantics as the ones present in the original Facade module. Finally, extracting this Bedrock code to x86 assembly yields executable binaries.

2.3 Issues with traditional extraction

The task that we describe in this thesis is very close to the problem of *extraction*. Traditional extraction is an unverified procedure that unsoundly inspects programs expressed in the logic of a proof assistant and produces a compilable program in an often closely related language. In Coq (Letouzey 2003), an OCaml plugin for extraction can be used to derive executable OCaml programs from Gallina sources. Although extraction was used in the original Fiat paper to obtain executable programs, it is not sufficient for our purposes:

- **Extraction is unverified**. Though there exists a pen-and-paper proof (Letouzey 2004) that the extraction logic is sound, no machine-checked proof has been written to establish the correctness of the corresponding implementation. Additionally, no verified compiler exists for any of the languages that extraction targets (mainly OCaml, though Haskell or Scheme code can also be obtained with a little bit of additional effort).

- **Extraction erases propositions and proofs**. This is very convenient in general, but it works against Fiat's encoding of computations as functions returning propositions: this encoding causes extraction to discard most Fiat programs (as demonstrated earlier in figure 1.3). Thus, to use standard extraction, Fiat refinements must bottom out into fully refined programs, instead of leaving bits of nondeterminism where linking against external data structures with non-fully-deterministic specifications could be useful.

- **Extraction is not soundly extensible**. Any optimizing transformation must be done either on the original Gallina terms or (unsoundly) on the resulting OCaml program. Though OCaml offers a rich programming environment covering both functional and imperative styles, there are no readily available mechanized semantic of OCaml to express transformations in. Additionally, as argued earlier in this thesis, certain transformations are best expressed and verified (for reasoning simplicity) as part of the extraction itself. The Coq to OCaml extraction engine provides the embryo of such a customization facility through its `Extract Constant` and `Extract Inductive` commands, but these mechanisms are unchecked, error-prone, and potentially unsound.

- **Extraction is single-language**. To preserve soundness, data structures used by Gallina programs must be written in Gallina as well and extracted along with the original programs. This prevents Gallina programs from depending on separately compiled, verified data structures written in other languages (such as imperative languages or assembly).

The next chapter presents an overview of our technique.

Chapter 3

Overview

Chapter 4 presents an in-depth account of our technique. Before diving in, however, it is useful to present the general intuition on a concrete example. We illustrate the compilation process on a simple example, starting with the following Fiat program p, simulating a die roll with 0 indicating the failing case:

```
p ≜ r ← Any
    ret (if r ≺ 7 then r else 0)
```

This program samples a random number using a call to an external function and returns its absolute value (≺ 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 (in this section, to help with understanding, slanted variables such as *prog* live in Facade land, underlined variables such as comp are Fiat computations, and regular variables such as p are Gallina terms; these annotations are omitted in the rest of this document):

- *p*, when started in a blank state (no variables defined, written ∅), 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 Fiat program $\underline{\mathtt{p}}$ shown above.

We write $\varnothing \overset{p}{\leadsto} [\![\ \text{`"out"} \twoheadrightarrow \underline{\mathtt{p}} \ ]\!]$ 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 $\underline{\mathtt{p}}$ in the variable `out`." One can think of it as a typical Hoare triple, with additional constraints on the pre- and postconditions to facilitate synthesis. Indeed, instead of the usual context where Hoare triples are used to help ensure that a program conforms to its specification, we use them here to drive the synthesis process; putting additional constraints helps us avoid unrealizable postconditions ($\perp$) and helps the compiler synthesize the Facade program. The precise definition shown in chapter 4 is constructed in such a way that Facade's semantics guarantee that any Facade 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 ≜ std.rand()
comp ≜ tmp < 7
If comp Then
  out ≜ tmp
Else
  out ≜ 0
EndIf
```

Expanding the definition of $\underline{\mathtt{p}}$, we find that we need to find a program p such that

$$\varnothing \overset{p}{\leadsto} \left[\!\!\left[\ \text{`"out"} \twoheadrightarrow \begin{array}{l} \mathtt{r} \leftarrow \underline{\mathtt{Any}} \\ \mathtt{ret}\ (\mathtt{if}\ \mathtt{r} \prec 7\ \mathtt{then}\ \mathtt{r}\ \mathtt{else}\ 0) \end{array} \right]\!\!\right]$$

We use our first *compilation lemma*, a variant of the *bind* rule (Figure 3.1b) to connect the semantics of Fiat's bind operation (the ← operator of monads) to the meaning of $\rightsquigarrow$, which yields the following synthesis goal:

$$\varnothing \overset{p}{\rightsquigarrow} [\![\ \texttt{`"tmp"} \leadsto \underline{\texttt{Any}}\ \texttt{as r}\ ]\!] :: [\![\ \texttt{`"out"} \leadsto \texttt{ret (if r < 7 then r else 0)}\ ]\!]$$

In this step, we have broken down a Fiat-level bind (`r` ← Any; …) assigned to a single variable into two parts, assigned to two variables: `tmp`, holding the result of the call to Any, and `out`, holding the final result. The double square-bracket notation describes states where the first variable `tmp` can be bound to any of the variables allowed by the call to Any, and where the second variable has one of the values allowed by the addition operation: although the addition is deterministic, it depends on the value returned by Any, bound as `r` at the Gallina level (the double-colon operator (`::`) is syntactic sugar for the consing operation on Fiat states). 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 computation Any 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:

$$\varnothing \overset{p_1}{\rightsquigarrow} [\![\ \texttt{`"tmp"} \leadsto \underline{\texttt{Any}}\ \texttt{as r}\ ]\!]$$

$$[\![\ \texttt{`"tmp"} \leadsto \underline{\texttt{Any}}\ \texttt{as r}\ ]\!] \overset{p_2}{\rightsquigarrow} [\![\ \texttt{`"tmp"} \leadsto \underline{\texttt{Any}}\ \texttt{as r}\ ]\!] :: [\![\ \texttt{`"out"} \leadsto \texttt{ret …}\ ]\!]$$

Notice the similarity between the specification of p_1 and the original specification of p: just like p moved from a blank state to a state binding `out` to a Fiat expression, p_1 moves from a blank state to a state binding `tmp` to a Fiat expression. At this point, a lemma about Facade's semantics tells us that `tmp` ≜ `std.rand()` is a good choice for p_1 (this is a *call* *rule* for `std.rand`; a lemma, presumably written by the author of the `std.rand` function, that connects its low-level semantics to that of the high-level Fiat construct $\underline{\texttt{Any}}$). We are therefore only left with p_2 to synthesize: noticing the common prefix of the starting and ending states, we apply the *chomp rule*, transforming the problem into the following:

$$\forall\ \texttt{r},\ \underline{\texttt{Any}} \leadsto \texttt{r} \rightarrow$$
$$\varnothing \xrightarrow[\texttt{["tmp"} \,\triangleright\, \texttt{r]}]{p_2} [\![\ \texttt{`"out"} \rightsquigarrow \texttt{ret (if r} \prec \texttt{7 then r else 0)}\ ]\!]$$

The additional mapping pictured under the $\rightsquigarrow$ arrow expresses an extra assumption that we can make about the starting and ending states: they must both map `tmp` to the same value `r` (the notation `[k ▷ v]` is a compact notation for an unordered map containing a single binding `(k, v)`). 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 $\underline{\texttt{Any}}$[1]. In this form, the synthesis goal matches the conclusion of the `If` compilation lemma: given three programs which respectively set a comparison variable $\texttt{v}_{\texttt{cmp}}$ to `1` if `r ≺ 7` and `0` otherwise (p_{Test}), compile the true branch of the conditional (p_{True}), and compile the false branch of the conditional (p_{False}), the following Facade program obeys the specification above:

```
pTest
If vcmp  == $1 Then
  pTrue
Else
  pFalse
EndIf
```

This gives us three new synthesis goals, which we can handle recursively. The three rules that we used are summarized in figure 3.1.

[1] 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: p_2 is an existential variable whose context does not contain `r`; it cannot depend on it.

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

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

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

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

$$\frac{\begin{array}{c} \varnothing \overset{p_{Test}}{\underset{\mathtt{ext}}{\rightsquigarrow}} [\![\ \grave{}\,\mathtt{v_{cmp}} \rightsquigarrow \mathtt{ret}\ \mathtt{p_{Test}}\ ]\!] \\ \mathtt{p_{Test}} \implies \varnothing \overset{p_{True}}{\underset{[\mathtt{v_{cmp}} \vartriangleright \mathtt{p_{Test}}]\ ::\ \mathtt{ext}}{\rightsquigarrow}} [\![\ \mathtt{k} \rightsquigarrow \underline{\mathtt{p_{True}}}\ ]\!] \\ \neg\ \mathtt{p_{Test}} \implies \varnothing \overset{p_{False}}{\underset{[\mathtt{v_{cmp}} \vartriangleright \mathtt{p_{Test}}]\ ::\ \mathtt{ext}}{\rightsquigarrow}} [\![\ \mathtt{k} \rightsquigarrow \underline{\mathtt{p_{False}}}\ ]\!] \end{array}}{\varnothing \overset{\begin{array}{l} p_{Test} \\ \mathtt{If}\ \mathtt{v_{cmp}}\ \mathtt{==}\ \$1\ \mathtt{Then}\ p_{True}\ \mathtt{Else}\ p_{False}\ \mathtt{EndIf} \end{array}}{\underset{\mathtt{ext}}{\rightsquigarrow}} [\![\ \mathtt{k} \rightsquigarrow \mathtt{if}\ \mathtt{p_{Test}}\ \mathtt{then}\ \underline{\mathtt{p_{True}}}\ \mathtt{else}\ \underline{\mathtt{p_{False}}}\ ]\!]}\ \textsc{If}$$

(c) The *if* rule: provided that the three intermediate programs p_{Test}, p_{True}, p_{False} respectively evaluate the condition of the if (assigning `1` to $\mathtt{v_{cmp}}$ for `true` and `0` for `false`), implement its true branch, and implement its false branch, we can connect the semantics of Facade's `If` statement to the Gallina-level `if`.

Figure 3.1: Three rules used by our compiler: *chomp*, *bind*, and *if*. In these rules, the notation $[\![\ \mathtt{k} \rightsquigarrow \underline{\mathtt{v}}\ ]\!]$ indicates a Fiat state biding mapping key `k` to Fiat computation $\underline{\mathtt{v}}$; $[\mathtt{k} \vartriangleright \mathtt{v}]$, on the other hand, indicates a Facade state binding mapping key `k` to Gallina value `v`.

Chapter 4

Technical description

This chapter describes our implementation in detail; when relevant, it points to the appropriate parts of our Coq development[1] (curious readers are encouraged to explore the code using this thesis as a guide for the implementation: the next chapter gives more details about the layout of the source tree).

At the core of our contribution is a relation (`ProgOk`) that connects Facade programs to pairs of Fiat states. Fiat states are ordered collections of named bindings, which can profitably be thought of as a compact and highly structured way to express pre- and post-conditions on Facade programs. To describe the full compilation process, we start by presenting the core definitions that allow us to connect Fiat and Facade programs through Fiat and Facade states, then discuss some of the crucial lemmas about these definition and how they are proved, detail how we translate nondeterministic choices to calls to externally specified functions, and finally present the details of our compilation logic and infrastructure.

[1]A snapshot of our development is tagged as `fiat-to-facade-MS-2016` in the Fiat repository at `https://github.com/mit-plv/fiat/`; setup instructions are in `src/CertifiedExtraction/README.rst`

4.1 Core definitions

4.1.1 Telescopes

We start by defining Fiat states, using a structure called a `Telescope`. Fiat states are concise and convenient ways to connect the semantics of Fiat and Facade: instead of expressing the pre- and postconditions that a Facade program must satisfy as a pair of arbitrary propositions, we collect all this information in a single data structure. The Coq definition of a telescope is given below (`av` is a type describing low-level ADTs known to Bedrock):

```
Inductive Telescope (av: Type) : Type ≜
| Nil : Telescope av
| Cons : ∀ T (key: NameTag av T)
           (val: Comp T)
           (tail: T → Telescope av),
    Telescope av.
```

As this definition makes clear, Fiat states have interesting properties that Facade states do not possess (recall that Facade states are unordered collections mapping variable names to either machine words, or heap-allocated ADTs described by the type `av`):

- They allow nondeterminism in values by storing computations instead of actual values. For example, it is legal to write `Cons `"k" (λ x ⇒ x > 1) Nil`.
- They do not directly distinguish between heap- and stack-allocated values, nor restrict values to Bedrock types; instead, they attach a "name tag" (Section 4.1.2) to each binding.
- They capture dependencies between variables by introducing an explicit ordering between them, allowing later bindings to depend on the values taken by previous ones.

In a sense, Fiat states are hybrids between Fiat programs and Facade states. As we will later see, a Fiat state is essentially isomorphic to a monadic program with names attributed to some of the bindings; thus, just like a Fiat program describes a set of acceptable return values, a Fiat state describes a set of acceptable environments (collections of names and values)—or, in other words, a set of acceptable Facade states.

To make telescopes easier on the eyes, we denote the telescope `Cons k v (λ w ⇒ t w)` as `⟦ k ⇝ v as w ⟧ :: t w`. Furthermore, when `v` is in the form `ret` v_0 we write `⟦ k ↦` v_0 `as w ⟧ :: t w` instead, and when the tail of the telescope does not depend on the bound variable `w` we omit the `as` clause: `⟦ k ↦ v ⟧ :: ...`.

4.1.2 NameTags

Each binding in a telescope comes with a name tag:

```
Inductive NameTag av T ≜
| NTNone : NameTag av T
| NTSome (key: string) (H: FacadeWrapper (Value av) T) : NameTag av T.
```

This tag describes which name refers to the associated value and how the value is represented in terms of low-level Bedrock types. Tags can be either of the following two values:

- `NTNone`, in which case the binding is anonymous and is simply a convenient way of expressing a monadic `Bind`; we might write, for example, something like the following to describe the Fiat program `x ← Any; ret x + x` (inside telescopes, we abbreviate `NTSome k _` by writing `` `k ``, and `NTNone` by stripping the arrow)

```
    ⟦ Any as x ⟧ :: ⟦ `"out" ↦ x + x ⟧ :: Nil
```

- `NTSome`, in which case the tag carries a name (as a string) and an instance of a type class describing how the associated value should be represented in terms of Bedrock types. This injection (a `FacadeWrapper` instance—section 4.1.3), is crucial to connect Fiat states to Facade states: it would be inconvenient to restrict Fiat programs to manipulating Bedrock types, but Facade programs have no knowledge of the complex types used at the Fiat level—instead, they think in terms of Facade's `Value` type, which allows either machine words (`W`) or low-level ADTs (encoded in a user-supplied type `av`):

```
Inductive Value {av} ≜
| SCA : W → Value
| ADT : av → Value.
```

To give an intuition of why these name tags are important, we return briefly to the definition of Telescopes. As previously mentioned, each cons cell of a telescope contains a computation, and a tail represented as a function mapping values permitted by the head to telescopes. This is a mixed-embedding analogue of a monadic `Bind`, where the consing operation plays the deeply embedded role of the shallow monadic binding. For this analogy to hold, though, we need the tail function to be passed a value of precisely the type of the head. This essentially prevents us from storing in telescopes only the low-level representation of values: since telescope tails compute on Fiat-level types, we cannot pass them low-level Bedrock encoding of these types (even less so given that Bedrock's encoding collapses all types into a single sum type).

4.1.3 FacadeWrappers

Injections from arbitrary Gallina types into low-level Bedrock types can be conveniently described by packaging them into instances of a FacadeWrapper type class (the importance of the injectivity requirement will become apparent when we discuss the relation used to connect Fiat and Facade states):

```
Class FacadeWrapper (WrappingType WrappedType: Type) ≜
{ wrap: WrappedType → WrappingType;
  wrap_inj: ∀ v v', wrap v = wrap v' → v = v' }.
```

4.1.4 WeakEq

The last missing element before we can connect Fiat and Facade states is a way to capture Facade's memory-tracking constraints: while Facade allows overwriting scalars and does not require them to be deallocated before returning control to the caller, it imposes much stronger restrictions on heap-allocated values. Accordingly, we define three relations on Facade states:

- `SameADTs` ensures that two Facade states have exactly the same heap-allocated variables:

```
Definition SameADTs {av} m₁ m₂ ≜
  ∀ k v, MapsTo k (@ADT av v) m₁ ↔
         MapsTo k (@ADT av v) m₂.
```

- `SameSCAs` ensures that the scalar bindings of one Facade state are all present in a second Facade state:

```
Definition SameSCAs {av} m₁ m₂ ≜
  ∀ k v, MapsTo k (@SCA av v) m₁ →
         MapsTo k (@SCA av v) m₂.
```

- `WeakEq` combines these two predicates:

```
Definition WeakEq {av} m₁ m₂ ≜
  @SameADTs av m₁ m₂ ∧ @SameSCAs av m₁ m₂.
```

4.1.5 SameValues and TelEq

The preceding definitions are enough to give the details of the relation connecting Fiat and Facade states. The recursive `SameValues` predicate expresses the fact that a particular Facade state is allowed by a given Fiat state, augmented for convenience with an unordered collection of deterministic bindings (this unordered collection will prove useful to describe bindings that are unmodified throughout a program's execution):

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

This relation expresses the following: a Facade state is related to a given Fiat state if there exists a sequence of nondeterministic choices such that all bindings of the Fiat telescope (augmented with the unordered collection of bindings) are exactly reflected in the Facade state, and any left-over bindings are only to stack-allocated values.

This ensures that:

- The Facade state's heap-allocated values are exactly the low-level encodings ("wrappings") of the values found along one of the nondeterministic paths in the Fiat telescope augmented with the unordered collection `ext`.

- The Facade state's stack-allocated values are a superset of the low-level encodings ("wrappings") of the values along the same nondeterministic path in the Fiat telescope augmented with `ext`.

Note how the injectivity requirement comes into play: all that we can learn from a `SameValues` instance is an equality between two wrapped values (`v = wrap w`, where `v` is extracted from `fmap` and thus is most often itself a `wrap` $\texttt{v}_0$). To be able to substitute $\texttt{v}_0$ for `w`, in the call to `tail w`, we need to be able to learn $\texttt{v}_0$ `= w` from `v = wrap w`, i.e. from `wrap` $\texttt{v}_0$ `= wrap w`; the other alternative being to require each telescope tail to be paired with a proof that the corresponding function is a morphism for the particular wrapping function used in the same cons cell (in other words, one could require the results returned by the tail function to depend only on the value of `wrap v`, regardless of the specific value of `w`—which amounts to `∀ w w', wrap w = wrap w' → tail w = tail w'`).

We write `fmap ≲ tenv ∪ ext` to make developments more readable. The following examples show how `SameValues` behaves, starting with valid examples:

- `["x" ▷ 1] :: ["y" ▷ (1,1)] ≲`
  `⟦ `"y" ↦ (1,1) ⟧ ∪ ["x" ▷ 1]`
- `["x" ▷ 1] :: ["y" ▷ (1,1)] ≲`
  `⟦ `"x" ↦ 1 ⟧ :: ⟦ `"y" ↦ (1,1) ⟧ ∪ ∅`
- `["x" ▷ 1] :: ["y" ▷ (1,1)] ≲`
  `⟦ `"x" ⇝ Any as x ⟧ :: ⟦ `"y" ↦ (x,x) ⟧ ∪ ∅`
- `["x" ▷ 1] :: ["y" ▷ (1,1)] :: ["z" ▷ 2] ≲`
  `⟦ `"x" ⇝ Any as x ⟧ :: ⟦ `"y" ↦ (x,x) ⟧ ∪ ∅`
  (We lost track of the `["z" ▷ 2]` binding; since 2 is a scalar, that is not an issue.)

On the other hand, here are some invalid examples:

- `["x" ▷ 1] ≴`

  `⟦ `"y" ↦ (1,1) ⟧ ∪ ["x" ▷ 1]`

  (`"y"` is missing; this is an issue because `(1,1)` is an instance of the pair ADT, not a scalar.)

- `["x" ▷ 1] :: ["y" ▷ (1,2)] ≴`

  `⟦ `"x" ⇝ Any as x ⟧ :: ⟦ `"y" ↦ (x,x) ⟧ ∪ ∅`

  (The value of `"y"` is not consistent with `(x,x)`.)

- `["x" ▷ 1] :: ["y" ▷ (1,1)] :: ["z" ▷ (1,2)] ≴`

  `⟦ `"x" ⇝ Any as x ⟧ :: ⟦ `"y" ↦ (x,x) ⟧ ∪ ∅`

  (We lost track of the `["z" ▷ (1,2)]` binding; since `(1,2)` is encoded as an ADT value, that is an issue.)

The `SameValues` relation can conveniently be used to define an equivalence relation on telescopes, `TelEq`:

```
Definition TelEq {A} ext (T₁ T₂: Telescope A) ≜
  ∀ st, st ≲ T₁ ∪ ext ↔
        st ≲ T₂ ∪ ext.
```

Due to this definition, any relation that only manipulates Fiat states through `SameValues` will automatically be a morphism for `TelEq`.

4.1.6 ProgOk

All of these definitions lead us to the final correctness condition that we base the synthesis process on. `ProgOk` summarizes the behavior of a Facade program in terms of two Fiat states `initial_tstate` and `final_tstate` both augmented with the same unordered collection of bindings `ext`:

```
Definition ProgOk {av} ext env prog
```

```
    (initial_tstate final_tstate: Telescope av) ≜
  ∀ initial_state: State av,
    (initial_state ≲ initial_tstate ∪ ext) →
    Safe env prog initial_state ∧
    ∀ final_state: State av,
      @RunsTo _ env prog initial_state final_state →
      (final_state ≲ final_tstate ∪ ext).
```

We write `{ t₁ } prog { t₂ } ∪ { ext } // env` in code and $\mathtt{t}_1 \overset{\mathtt{prog}}{\underset{\mathtt{ext}}{\rightsquigarrow}} \mathtt{t}_2$ in text, to mean `ProgOk ext env prog t₁ t₂` (we usually omit `env` in text); this Hoare-triple notation captures the fact that running the program in any of the Facade states described by the precondition is safe and that, if the program terminates, it yields one of the states allowed by the postcondition. Additionally, this notation makes explicit the set of external methods that the program being refined may use (`env`).

4.2 Supporting lemmas and proof automation

Each of the aforementioned definitions enjoys specific properties, in particular in connection to the relations that it is built from. The next subsections describe some of these interesting properties and how they are established. Most of the corresponding proofs are written with a high degree of automation; in general, they can be classified by how deep they dive in the stack of nested definitions, as explained below.

4.2.1 FacadeWrapper

The injection relation used to describe encodings of Fiat objects into Bedrock ADT values is transitive (injections from A to B and B to C can be composed to form an injection from A to C) and reflexive (the identity function injects a type in itself). These two properties allow us to define a large number of such injections: we inject simple types into Facade scalars (32-bit stack-allocated machine words) and more complex types into Bedrock's type of heap-allocated values.

Most injectivity proofs, when Fiat types conveniently line up with the Bedrock types in which they are encoded, are very simple. In some cases, however, the encoding is more complex: for example, encoding bounded numbers (`{n | n < pow`$_2$` 32}`) as machine words entails proving unicity of comparison proofs (i.e. proving that any fact of the form `a < b` admits a single proof) or taking proof irrelevance as an axiom.

4.2.2 WeakEq

The `WeakEq` relation is transitive, reflexive, and is a morphism for equality of Facade states (which itself is defined in terms of bi-directional inclusion of the set of pairs of keys and values). Among many other properties, we prove that `WeakEq` is preserved by symmetric additions to and removals from both states and that various functions used by Facade on states, such as the one checking whether a given variable maps to an ADT value in a given state, are covariant morphism for weak equality of states.

4.2.3 SameValues and TelEq

The `SameValues` relation is a morphism for various relations (more precisely, if `TelEq` t_1 t_2 then $st \lesssim t_1 \cup ext \leftrightarrow st \lesssim t_2 \cup ext$ for any `st` and `ext`; and if `WeakEq st st'` and `WeakEq ext' ext` then $st \lesssim t \cup ext \rightarrow st' \lesssim t \cup ext'$). `TelEq`, on the other hand, is an equivalence relation on Fiat states. Properties of both of these relations are proven in `CoreLemmas.v` and `ExtendedLemmas.v` and used as building blocks in proofs of `ProgOk` statements. In particular, many proofs about `SameValues` connect the semantics of Fiat states to that of Facade programs; one of the most important such properties concerns Fiat's bind operation; this rule, demonstrated previously, allows us to translate Fiat's anonymous binds into named ones, as cons cells in telescopes:

```
Lemma SameValues_Fiat_Bind_TelEq :
  ∀ {av A B} (key: NameTag av B) compA compB tail ext,
    TelEq ext
          ⟦ key ⇝ (a ← compA; compB a) as ab ⟧ :: tail ab
          ⟦ compA as a ⟧ :: ⟦ key ⇝ compB a as ab ⟧ :: tail ab.
Proof.
  unfold TelEq; SameValues_Fiat_t.
Qed.
```

Most proofs of this type are discharged by a single tactic `SameValues_Fiat_t`, which unfolds `SameValues` and simplifies the resulting expressions using available information about the Facade state that `SameValues` is applied to.

4.2.4 ProgOk

Proofs about `ProgOk` are the most numerous. They fall roughly into four categories:

- Proofs about the semantics of `ProgOk`, such as `ProgOk` being a morphism for `TelEq`, various properties about allocating and forgetting about of scalars, and most importantly variants of the *chomp* rule (Figure 3.1a):

```
     Lemma ProgOk_Chomp_Some :
       ∀ `{FacadeWrapper (Value av) A} env key value prog
         (tail₁ : A → Telescope av) (tail₂ : A → Telescope av) ext,
         key ∉ ext →
         (∀ v: A, value ⤳ v →
                  { tail₁ v }
                    prog
                  { tail₂ v } ∪ { [key ▷ wrap v] :: ext } // env) →
         { ⟦ `key ⇝ value as v ⟧ :: tail₁ v }
           prog
         { ⟦ `key ⇝ value as v ⟧ :: tail₂ v } ∪ { ext } // env.
```

- Proofs about the connection between Fiat's monadic semantics and `ProgOk` itself, and most importantly the *bind* rule; thanks to `ProgOk` only manipulating telescopes through the `SameValues` relation, and thus being a morphism for `TelEq`, these proofs most often can be recast in terms of `TelEq` and `SameValues` — as is the case for the *bind* rule, which was listed above. This rule, phrased in terms of `ProgOk`, is shown in figure 3.1b; but by proving that `ProgOk` is a morphism for telescope equality, we can simply use the setoid machinery (Sozeau 2009) to rewrite with the corresponding `TelEq` lemma, without materializing a `ProgOk` one.

- Proofs about the connection between Facade's operational semantics and `ProgOk` itself; these proofs are essentially recastings of Facade's operational semantics in terms of telescopes and seldom need to unfold the `SameValues` relation. One special subclass of these rules is call rules, which are important enough to deserve their own section (4.2.5); other ones include simple connections such as `CompileRead`:

```
Lemma CompileRead:
  ∀ {av} name var (val: W) ext (tenv: Telescope av),
    name ∉ ext →
    NotInTelescope name tenv →
    MapsTo var (wrap val) ext →
    ∀ env,
    { tenv }
      name ≜ var
    { ⟦ `name ↦ val as _ ⟧ :: tenv } ∪ { ext } // env.
Proof.
  SameValues_Facade_t.
Qed.
```

- Proofs of complex optimizations translating Gallina patterns into lower-level operations, such as our motivating example of translating folds into destructive iteration over mutable lists.

Most of these proofs are discharged using a single tactic `SameValues_Facade_t`; the last category, however, is often a combination of such automation and manual proofs of the most tricky parts of the optimization.

4.2.5 Call rules

One particularly interesting class of proofs, and arguably one of the places where this new synthesis-based compilation technique shines, is *call rules*. Facade's semantics allow calls to axiomatically specified methods, guaranteeing that, as long as the method's preconditions are satisfied, the method's postconditions will hold in the post-call state.

This section describes how such calls are introduced in the compilation process.

Bedrock specifications

In Facade, calls to externally implemented methods are made by name and matched at compile time to axiomatic specifications stored in an environment of external functions exposed to the program. Facade method specifications have the following form (`Value` is the type that Facade uses to represent either machine words (`W`) or ADTs (`av`)):

```
Inductive Value {av} ≜
| SCA : W → Value
| ADT : av → Value.

Record AxiomaticSpec {av} ≜
{ PreCond (input : list Value) : ℙ;
  PostCond (input_output : list (Value * option av)) (ret : Value) : ℙ;
  PreCondTypeConform : type_conforming PreCond }.
```

Preconditions are expressed as propositions on lists of arguments; postconditions as propositions on return values and lists of pairs of original arguments and (when the argument is an ADT value, thus passed by reference) post-call values. The last field of the `AxiomaticSpec` record is a sanity check ensuring that arguments are consistently typed;

that is, the precondition must at least ensure that each argument is either consistently an ADT value or consistently a scalar.

In general, pre- and postconditions follow a very regular form: they assert the existence of a number of parameters of certain types such that their input lists are lists of these parameters, and in the case of postconditions describe return values and mutated ADT values in terms of these same arguments. Additional, more complex side conditions are at times present.

As a concrete example, here is the specification of the `Tuple[W].put` method of a type of arrays of machine words (called “tuple” later in this thesis; more details about this ADT are given in section 6.3.5).

```
Definition Put : AxiomaticSpec ADTValue.
  refine {|
      PreCond ≜ λ args ⇒
        ∃ ls pos val,
          args = [ADT (TupleConstructor ls); SCA _ pos; SCA _ val] ∧
          pos < natToW (length ls);
      PostCond ≜ λ args ret ⇒
        ∃ ls pos val,
          args = [(ADT (TupleConstructor ls),
                   Some (TupleConstructor (Array.upd ls pos val)));
                  (SCA _ pos, None); (SCA _ val, None)] ∧
          ret = SCA _ 0
    |}; crush_types.
Defined.
```

The definition of the `List[W].push` method of lists of machine words is very similar:

```
Definition Push : AxiomaticSpec ADTValue.
  refine {|
      PreCond ≜ λ args ⇒
        ∃ l w,
          args = [ADT (WordList l); SCA _ w];
      PostCond ≜ λ args ret ⇒
        ∃ l w,
          args = [(ADT (WordList l),
                   Some (WordList (w :: l)));
                  (SCA _ w, None)] ∧
          ret = SCA _ 0
    |}; crush_types.
Defined.
```

Connecting Bedrock and Fiat specifications

These specifications are in most cases not usable as-is to compile Fiat programs: they are phrased in terms of low-level types and do not match exactly the operations modeled in Fiat land using nondeterministic choices. We can, however, lift most of these specifications relatively easily to make them more readily usable. For this, we recast them in terms of telescopes: the transformation is made on a case-by-case basis, recasting pre- and post-conditions in terms of high-level Fiat types and reformulating these specifications in terms of telescope-based characterizations of the semantics of Facade calls to these methods.

Concretely, a call rule for the aforementioned `List[W].push` method may look like the following (`(PreconditionSet tenv ext vars)` is a shorthand to assert that all variables in `vars` are distinct and that none of them are in either `tenv` or `ext`)

```
Lemma CompileWordList_push_spec :
∀ vhd  vlst  fPush env ext tenv h (t: list W),
  MapsTo fPush (Axiomatic WordListADTSpec.Push) env →
  PreconditionSet tenv ext [vhd ; vlst ] →
  { ⟦ `vlst  ↦ t as _ ⟧ :: ⟦ `vhd  ↦ h as _ ⟧ :: tenv }
    call fPush(vlst , vhd )
  { ⟦ `vlst  ↦ h :: t as _ ⟧ :: tenv } ∪ { ext } // env.
```

Proofs of such properties are generally automated, with some hints provided through rule-specific lemmas when the correspondence between Fiat and Facade specifications is not entirely straightforward. The basic `SameValues_Facade_t` takes care of most of the proofs, and a specialized tactic `facade_cleanup_call` is used to handle hypotheses inherited from the semantics of `RunsTo` on calls. Specialized hand-crafted versions of the lemmas are sometimes exported, with some limited amount of manual work needed to establish the correspondence with the automatically proven rule.

Writing call rules

Writing call rules presents library authors with an interesting decision space. For example, one can try to write rules as closely mirroring the original Facade specifications as possible and leave it to compilation tactics to transform compilation goals until they match this faithful-to-the-original form. Conversely, one can identify specific points in a compilation run where a particular call rule may be useful and prove rules tailored for this.

Designing the right call rules amounts to deciding how much work should be accomplished dynamically by tactics during derivations and, conversely, how much insight about derivations should be encoded directly in lemmas. The following list summarizes some of the main decisions:

- **Single-step versus multi-step**: call rules can either be written under the assumption that method arguments will already have been made available or not. In the first case, call rules will only mention the call to the external method; in the second one, the will explicitly mention multiple subprograms, each tasked with allocating the right arguments.

  Concretely, one can either write the following, in which case the caller will have some amount of work to do to ensure that h is indeed available at call time:

```
Lemma CompileWordList_push_spec :
∀ v_hd  v_tl  fPush env ext tenv h (t: list W),
  MapsTo fPush (Axiomatic WordListADTSpec.Push) env →
  PreconditionSet tenv ext [v_hd ; v_tl ] →
  { ⟦ `v_tl  ↦ t as _ ⟧ :: ⟦ `v_hd  ↦ h as _ ⟧ :: tenv }
    call fPush(v_tl , v_hd )
```

```
{ ⟦ `vtl ↦ h :: t as _ ⟧ :: tenv } ∪ { ext } // env.
```

Or one could write this, which naturally exports a side goal requiring to produce a program `pArg` constructing or loading `h`:

```
Lemma CompileWordList_push_spec :
∀ vhd vtl fPush env ext tenv h (t: list W),
  MapsTo fPush (Axiomatic WordListADTSpec.Push) env pArg →
  PreconditionSet tenv ext [vhd ; vtl ] →
  { ⟦ `vtl ↦ t as _ ⟧ :: tenv }
    pArg
  { ⟦ `vtl ↦ t as _ ⟧ :: ⟦ `vhd ↦ h as _ ⟧ :: tenv } ∪ { ext } // env →
  { ⟦ `vtl ↦ t as _ ⟧ :: tenv }
    pArg;
    call fPush(vtl , vhd )
  { ⟦ `vtl ↦ h :: t as _ ⟧ :: tenv } ∪ { ext } // env.
```

The second form is simpler to apply, at the cost of making the proof of the lemma redundant (the proof is essentially a combination of the proof regarding the semantics of sequencing with the first proof). The first form is trickier to apply and generally requires devising a slightly smarter tactic. Finally, one can also first prove the first form, then prove the second form from it.

- **Requiring values to be present in preconditions**: there are multiple ways to express that a telescope should contain a particular binding (for example an argument to the method being called): one can either make a specialized lemma in which the first cell of the precondition telescope is the desired binding, or one can instead express in generic terms the fact that the precondition telescope must contain that binding.

  Concretely, one can write one of the previous two forms (explicit preconditions) or instead write the following:

```
∀ vhd vtl fPush env ext tenv h (t: list W),
  MapsTo fPush (Axiomatic WordListADTSpec.Push) env →
  Lifted_MapsTo ext tenv v_hd  (wrap h) →
  Lifted_MapsTo ext tenv v_tl  (wrap t) →
  v_hd  ≠ v_tl  → v_hd  ∉ ext → v_tl  ∉ ext →
  { tenv }
    call fPush(v_tl , v_hd )
  { ⟦ `v_tl  ↦ h :: t as _ ⟧ :: DropName v_tl  tenv } ∪ { ext } // env.
```

This time a new predicate `Lifted_MapsTo` is used; it expresses that a property holds of any state that is `SameValues`-related to a pair of a telescope and an additional collection of bindings:

```
Definition LiftPropertyToTelescope
  {av} ext (prop: _ → ℙ) : Telescope av → ℙ ≜
    λ tenv ⇒ ∀ st, st ≲ tenv ∪ ext → prop st.

Definition Lifted_MapsTo {av} ext tenv k v ≜
  LiftPropertyToTelescope ext (MapsTo k v) tenv.
```

This form is particularly convenient, because the _ ≲ _ ∪ _ instance that we need to specialize a `LiftPropertyToTelescope` hypothesis is exactly what we get to assume when proving the safety part of a `ProgOk`. Then, to express the fact that v_{tl}'s binding has changed, we introduce a straightforward `DropName` function which removes a binding from a telescope (more precisely, it makes that binding anonymous):

```
Fixpoint DropName {A} needle (haystack: Telescope A) ≜
match haystack with
| Nil ⇒ Nil
| Cons T nt val tail ⇒
  Cons (match nt with
        | NTSome key H ⇒ if string_dec key needle then NTNone else nt
        | NTNone ⇒ nt
        end) val (λ v ⇒ DropName needle (tail v))
end.
```

A variation on this theme, which requires lighter machinery, is given by the following example. This form is very close to the original one, but it gives a stronger hint to the user of the lemma as to what to do next in the proof: instead of rewriting the precondition of the telescope so as to make it match the lemma's formulation *before* applying the lemma (as was done in the very first example), the user can now directly apply the lemma and then simply prove that the precondition had the right shape:

```
∀ vhd  vtl  fPush env ext tenv tenv' h (t: list W),
  MapsTo fPush (Axiomatic WordListADTSpec.Push) env →
  PreconditionSet tenv ext [vhd ; vtl ] →
  TelEq tenv ([[ `vtl  ↦ t as _ ]] :: [[ `vhd  ↦ h as _ ]] :: tenv') →
  { tenv }
    call fPush(vtl , vhd )
  { [[ `vtl  ↦ h :: t as _ ]] :: tenv' } ∪ { ext } // env.
```

In general, it is convenient to write things in the second or third form and derive a specialized rule in the third form. Using rules in the third form, however, does require some amount of setoid rewriting to establish the `TelEq` proof, which tends to make things slow.

- **Dropping bindings**: Similarly, if a call deallocates a variable, one can either explicitly mention it in the precondition and drop it from the postcondition, or one can define a generic `DropName` function that removes a binding from a telescope and formulate the rules in terms of that predicate. Examples of this alternative are very similar to the previous item.

- **Including continuations**: In practice, pushing a value into a list is in most cases not the only operation that the program will have to accomplish: there will thus be further differences between the initial state and the final state than just this binding, and it will be necessary to introduce multiple sequencing steps in the final program. This introduction can be done by a tactic guessing which intermediate state the program will have at this intermediate point, or it can be done as part of a lemma; in that case, the lemma essentially enforces how the proof proceeds (and it can thus enforce reasoning in the weakest-precondition style, for example). Concretely, the `Push` lemma above could be extended as follows:

```
∀ vhd  vtl  fPush env ext tenv h (t: list W),
  MapsTo fPush pAfter  (Axiomatic WordListADTSpec.Push) env →
  PreconditionSet tenv ext [vhd ; vtl ] →
  { ⟦ `vtl  ↦ h :: t as _ ⟧ :: tenv }
    pAfter
  { ⟦ `vtl  ↦ h :: t as _ ⟧ :: tenv' } ∪ { ext } // env →
  { ⟦ `vtl  ↦ t as _ ⟧ :: ⟦ `vhd  ↦ h as _ ⟧ :: tenv }
    call fPush(vtl , vhd )
    pAfter
  { ⟦ `vtl  ↦ h :: t as _ ⟧ :: tenv' } ∪ { ext } // env.
```

- **Copying arguments**: A scalar argument of a function might already be present in the context in some cases, whereas it might require allocating in others. One can choose to prove specialized copies of the lemma. The first one would be applicable when the variable is present and pass it directly to the function being called, whereas the second one would be applicable when it is not present and would create an appropriate temporary to pass to the function being called (in fact, one can even subdivide the first case into checking whether the variable is present in the telescope and checking whether it is present in the extended collection of bindings). On the other hand, one can also prove a single, generic lemma that always requires a temporary to be created; finally, one can prove a single lemma that assumes that the variable is there, leaving it to compilation tactics to ensure that it indeed is. All of these patterns are already demonstrated above.

It is not clear which form is generally the best, though a combination of making intermediate states explicit, proving complex lemmas in terms of `LiftPropertyToTelescope` and DropName, and forcing the copying of scalar arguments (these copies can then be optimized away by Facade's constant-folding pass) seems to work best, especially if an extra lemma is then added to express preconditions in terms of TelEq. We have automated tactics that serve as incomplete decision procedures for these TelEq side conditions (decide_TelEq_instantiate), as well as helper tactics that extend SameValues_Facade_t to handle Lifted and DropName (Lifted_t).

4.3 Compilation infrastructure

With these definitions in place and these lemmas proven, we are ready to start compiling programs.

4.3.1 Setting up

Compilation tasks are phrased in terms of the ProgOk statement; that is, our main compilation tactic takes a goal whose consequent is a ProgOk and progressively compiles the postcondition by breaking it down into smaller subgoals of the same ProgOk form. At the very beginning, however, our goals are not exactly in ProgOk form: instead, they are in the form of a Σ-type (a pair of values in which the type of the second element depends on the value of the first one) whose first element is a program and whose second element is a proof that that program satisfies a proposition in the form of a ProgOk (possibly quantified by multiple input variables). The following is an example of such a goal:

```
{prog : Stmt &
 ∀ x y : W,
 { ⟦ `"x" ↦ x ⟧ :: ⟦ `"y" ↦ y ⟧ :: Nil }
    prog
 { ⟦ `"out" ↦ x ⊕ y ⟧ :: Nil } ∪ { ∅ } // env}
```

Crucially, the existential is at the very top of the formula; our first step is to instantiate it with an existential variable whose context is essentially empty. We are thus forced to produce a single program, which may never depend on the values of intermediate variables introduced during compilation.

4.3.2 Basic blocks

We start by implementing a basic tactic that perform minor cleanups: introducing variables if the goal is quantified, creating the synthesized program's existential variable if it has not yet been created, substituting equalities if any are present in the context, etc.

We then define simple tactics to handle basic Gallina constructs: conditionals, arithmetic, Boolean tests, and constants. These tactics mostly apply straightforward lemmas, but the arithmetic and conditionals ones do a bit of work to translate Gallina-level operations on machine words into Facade's operators. For simplicity, we do not produce complex Facade expressions; instead, we recursively break down compound arithmetic expressions. Concretely, this means that for `(x + (y - 1))` we produce a temporary variable for `y - 1` and then add it to `x`:

```
tmp ≜ y - 1
out ≜ x + tmp
```

Fortunately, the Facade compiler is smart enough to (essentially) optimize the temporary away.

The rest of our tactics mix more complex optimizations and compilation patterns with handling of Fiat's nondeterministic semantics:

- We transform uses of Fiat's bind operators into individual telescope cells, using the *bind* rule, as demonstrated in chapter 3.
- When the head bindings of the pre- and postconditions match we apply the *chomp* rule to simplify the synthesis goal. Similarly, when we can find matching bindings

deep down in the pre- and postconditions, we also heuristically try to move them to the front to apply the *chomp* rule, using setoid rewriting.

- When we notice that one of the variables in the precondition is not used in the postcondition and we have access to a destructor for its Bedrock type, we deallocate it.

- When we see a fold or a map we apply the corresponding loop compilation rule, which we now describe in more detail.

4.3.3 Compiling loops

Many of the programs that we compile involve folds over lists of values; additionally, in many cases, the list is not reused after iteration. This suggests the following optimization: instead of compiling `fold_left` as a recursive higher-order function then separately compiling the function being folded and finally combining both compiled implementations to implement the original loop, we can instead compile `fold_left` into a while loop (in a sense doing the analogue of tail call elimination) and then compile the folded function separately into a chunk of Facade code guaranteed to implement the behavior of the original function (except for the fact that the Facade implementation is required to mutate its input arguments). Since the while loop iterates over the source list destructively, the rule does not apply if the list is then reused later in the program (which, as a rough approximation, can be detected by looking for keys mapping to the list itself in the postcondition of the program being synthesized).

Figure 4.1 shows a deductive-style presentation of one of the loop compilation rules; the corresponding Coq code is a bit of a mouthful but still quite comprehensible (Figure 4.2). The generality of `fold_left` allows us to compile maps and reverses in the same way.

$$\forall\ \text{h a lst},$$

$$\frac{[\![v_{hd} \mapsto \text{h}]\!] :: [\![v_{ret} \rightsquigarrow \text{a}]\!] :: \text{t} \; \underset{[v_{lst} \rhd \text{wrap lst}] :: \text{ext}}{\overset{\text{pBody}}{\leadsto}} \; [\![v_{ret} \rightsquigarrow \text{f a h}]\!] :: \text{t}}{\begin{array}{l}[\![v_{ret} \rightsquigarrow \text{init}]\!] :: \\ [\![v_{lst} \mapsto \text{lst}]\!] :: \text{t}\end{array} \; \underset{\text{ext}}{\overset{\text{LOOP(pBody, lst)}}{\leadsto}} \; [\![v_{ret} \rightsquigarrow \text{fold\_left f lst init}]\!] :: \text{t}}\ \textsc{FoldL}$$

Figure 4.1: The *FoldL* rule, connecting the functional reduction of `lst` with `f` on initial value `init` and the imperative computation of the same value using destructive iteration on a mutable list. `LOOP` is a macro whose body expands as shown in figure 4.2.

```
Lemma CompileLoop {N} :
∀ {A} `{FacadeWrapper (Value QsADTs.ADTValue) A}
  lst init v_head  v_test  v_lst  v_ret  fPop fEmpty fDealloc facadeLoopBody
  env (ext: StringMap.t (Value QsADTs.ADTValue)) tenv
  (f: Comp A → FiatWTuple N → Comp A),

  MapsTo fPop (Axiomatic Pop) env →
  MapsTo fEmpty (Axiomatic Empty) env →
  MapsTo fDealloc (Axiomatic Delete) env →

  PreconditionSet tenv ext [v_head ; v_test ; v_lst ; v_ret ] →

  (∀ head (acc: Comp A) (s: list (FiatWTuple N)),
      { ⟦ `v_ret  ⇝ acc ⟧ :: ⟦ `v_head  ↦ head ⟧ :: tenv }
        facadeLoopBody
      { ⟦ `v_ret  ⇝ (f acc head) ⟧ :: tenv }
      ∪ { [v_test  ▷ wrap false] :: [v_lst  ▷ wrap s] :: ext } // env) →

  { ⟦ `v_ret  ⇝ init ⟧ :: ⟦ `v_lst  ↦ lst ⟧ :: tenv }
    is_empty ≜ fEmpty(seq)
    While (!is_empty)
        head ≜ fPop(seq)
        facadeLoopBody
        is_empty ≜ fEmpty(seq)
    EndWhile
    call fDealloc(v_lst )
  { ⟦ `v_ret  ⇝ (fold_left f lst init) ⟧ :: tenv } ∪ { ext } // env.
```

Figure 4.2: Lightly edited copy of the FoldL call rule, presented as a Gallina lemma.

4.3.4 Extending the compiler

Extensibility is one of the key aspects of our compilation strategy. We detail a number of simple ways in which the compiler's behaviors can be extended; in many cases, we make forward references to our benchmarks, in which more detailed examples are given.

- **Intrinsics**: instead of unfolding Gallina functions, we can prove them equivalent to low-level operations provided by Bedrock and compile them away. This is akin to using compiler intrinsics for better performance; it is demonstrated in the discussion of microbenchmarks (Section 6.1.4).
- **Rewrites**: It is often easy to apply rewrites to change the goal into a form that can be handled by existing tactics. For example, `fold_right` can be rewritten in terms of `fold_left` and `rev`. Here, too, we give a more detailed example as part of our microbenchmarks (Section 6.1.3).
- **Arbitrary new rules**: Compilation tactics are independent from each other; they can be ordered, but in general they tend to be specific enough to compose easily, without committing to a given ordering. The intuition is essentially that each tactic is predicated on sufficiently precise conditions that applying it should not lead into a dead end; thus we don't need backtracking, as we can simply add new tactics to handle new patterns.

  To make this task easier, we provide a number of helper tactics:

  - `match_ProgOk` recognizes a `ProgOk` goal and passes its individual components to a caller-supplied continuation.
  - `may_alloc(var)` checks whether a variable is already present in the precondition or if it can be created as a new variable (it helps prevent tactics from looping).

- `gensym(name)` constructs a Gallina string not appearing in either the goal or the hypotheses. It uses `name` as the string's prefix and appends a number to it to guarantee uniqueness.

- `move_to_front(var)` uses setoid rewriting to swap independent bindings and move a particular key to the front of a telescope.

- `compile_do_side_conditions` handles a number of common side conditions, including ensuring that variable names do not collide, finding a pointer to a specific axiomatic specification in the current environment of imports, ensuring that a name does not appear in a telescope, etc.

- `call_tactic_after_moving_head_binding_to_separate_goal` takes a compilation tactic and applies after introducing a carefully crafted intermediate state and running a number of sanity checks. This tactic, like other similar combinators, uses the lemma about Facade's sequencing semantics to introduce an intermediate program point. Applying these combinators properly is generally nontrivial; hence our earlier remarks (Section 4.2.5) about including such intermediate states in lemma statements instead.

The discussion of `Any` in section 6.1 gives an example of such an extension.

4.3.5 Cleaning up the resulting programs

Many of the aforementioned tactics suffer from one small drawback: they often introduce no-ops (`Skip`) in the final, compiled program. This is due to the way certain compilation lemmas are phrased: this commonly happens if a lemma introduces an intermediate state, which turns out to not be required. For example, a lemma about addition could break down compiling addition into three steps: placing the first argument into a variable `a`, the second one into a variable b, and finally assigning to the result the Facade sum of the two arguments. It could be, however, than at the particular point in compilation when the lemma is introduced these two variables a and b are already in context; in that case, by calling the lemma with the right parameters, the first two steps are no-ops.

Although later compilation stages are generally good at optimizing away these minor inefficiencies, it is aesthetically more pleasant to remove them before presenting the generated programs to users. For that purpose, we prove our own optimization, in the form of the following compilation rule (parts of its proof were developed by Peng Wang):

```
Lemma ProgOk_RemoveSkips {av} :
  ∀ (prog : Stmt) (pre post : Telescope av) ext env,
    { pre } prog { post } ∪ { ext } // env →
    { pre } RemoveSkips prog { post } ∪ { ext } // env.
Proof.
  unfold ProgOk; intros * ok ** sv; specialize (ok _ sv);
    intuition eauto using RemoveSkips_Safe, RemoveSkips_RunsTo.
Qed.
```

In this rule, RemoveSkips stands for the following simple recursive definition:

```
Definition Is_Skip : ∀ p, { p = Skip } + { p ≠ Skip }.
Proof. … Defined.

Fixpoint RemoveSkips (s: Stmt) ≜
  match s with
  | Skip ⇒ Skip
  | Seq a b ⇒ let a ≜ RemoveSkips a in
              let b ≜ RemoveSkips b in
                if (Is_Skip a) then b
                else if (Is_Skip b) then a
                else (Seq a b)
  | If c t f ⇒ If c (RemoveSkips t) (RemoveSkips f)
  | While c b ⇒ While c (RemoveSkips b)
  | Call r f args ⇒ Call r f args
  | Assign x v ⇒ Assign x v
  end.
```

Interestingly, this rule can be applied at any time: we apply it a single time, at the very beginning of compilation.

Chapter 5

Notes about the implementation

This section dives into more concrete implementation details: we describe the organization of the source code; give statistics about proofs, tactics, and theorems; and discuss the automation strategy used in constructing domain-specific (incomplete) decision procedures for the main classes of theorems that compose the framework. The code was written in Coq 8.4 (The Coq development team 2015) and can be run in any Coq IDE (we use Proof General (Aspinall 2000) with Company-Coq (Pit-Claudel and Courtieu 2016)). Our derivations are assumption-free in Coq's Calculus of Inductive Constructions augmented with Axiom K (Streicher 1993) and ensemble extensionality (a limited form of functional extensionality). Our trusted base is limited to:

- The host operating system's assembler
- The Coq proof checker (OCaml, 7000 lines)
- The translator from Bedrock to x86 assembly (Gallina, 200 lines)
- Hand-written assembly wrappers for extracted methods (x86, 50 lines)

5.1 Organization of the Coq sources

```
./src/CertifiedExtraction
├── Benchmarks
│   │ Various compilation examples
│   │
│   ├── MicrobenchmarksAnnotated.v
│   │   Examples of small Fiat programs being compiled to Facade
│   ├── ProcessScheduler.v
│   │   This thesis' core example, compiling database queries
│   └── MicroEncoders.v
│       Another application domain, producing efficient packet serializers
│
├── Extraction
│   │ Implementation of the extraction logic
│   │
│   ├── External
│   │   Calls to generically defined data structures
│   │
│   ├── QueryStructures
│   │   │ Compilers for SQL-style queries
│   │   │
│   │   ├── CallRules
│   │   │   Calls to Bedrock data structures
│   │   └── Wrappers.v
│   │       Injections of Fiat data types into Bedrock
│   │
│   └── BinaryEncoders
│       │ Compilers for packet serialization code
│       │
│       ├── CallRules
│       │   Calls to Bedrock data structures
│       └── Wrappers.v
│           Injections of Fiat data types into Bedrock
│
├── ADT2CompileUnit.v
│   Translation from Fiat specs to Bedrock specs (B. Delaware)
├── Core.v
│   Main definitions: Telescopes, Hoare triples, Wrappers
└── …
    Properties of telescopes and triples
```

Figure 5.1: Organization of the Coq sources of this thesis

Figure 5.1 gives an overview of our code, found in a subdirectory of the Fiat repository. The main entry point is `Core.v`; other files of interest include `MicroBenchmarksAnnotated.v` and `ProcessScheduler.v` in `Benchmarks`. Many compilation rules can be found in the

`Extraction/QueryStructures/CallRules` folder, whereas most basic compilation blocks are directly in the `Extraction` folder.

Compilation instructions can be found in the `README.rst` file at the root of the `CertifiedExtraction` folder.

5.2 Highly automated proofs in SMT style

Most of the proofs in this thesis' supporting code are highly automated. They often follow similar patterns and fall in a few broad classes: writing them all manually, in fully expanded form, would be tedious and extremely time consuming[1]. The next paragraphs describe the style of automation that is used throughout the codebase.

5.2.1 The `learn` tactic

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

[1] `coqwc` reports a ratio of 7670 lines of specifications to 3250 lines of proofs, supporting the general claim that the proofs are highly automated (these counts are, however, approximate: `coqwc` tends to overestimate specification counts against proof counts).

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

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

This `learn` tactic elegantly solves the usual issue of having to decide when to specialize a fact, if at all. In our case, it most commonly happens with `ProgOk`: we may know some instance of `ProgOk`, and that instance can be specialized to produce a facts about any pair of initial and final Facade states. If we specialize it with the wrong states, however, the goal will become unprovable. Using `learn` to instantiate a copy of that fact, on the other hand, ensures that we do not weaken an hypothesis, that we do make progress, and that we do not get into a loop, repeatedly learning the same instantiation of the `ProgOk` fact.

5.2.2 Saturation proofs

The `learn` tactic, therefore, is useful doing saturation proofs: we can collect a large number of facts before using bottom-up reasoning (Coq's `eauto`) to conclude a proof. In a sense, it allows us to implement customizable generalizations of Coq's `intuition`: tactics can use large numbers of patterns to enrich the context with individual facts before deferring to Coq's `eauto` once the context is saturated (i.e. no new facts can be learnt by applying the tactics rules).

This style of proofs is convenient in our case, because many proofs consist of nontrivial manipulations of telescopes; by learning many facts about e.g. variants of `SameValues` with bindings added to the telescope, to the Facade state, or to the extra collection of unordered bindings, we greatly simplify proofs and make many goals easier to discharge automatically.

This technique is inspired by trigger-based instantiation of quantifiers used in SMT solvers; many SMT solvers indeed allow users to annotate their quantifiers with patterns that drive the SMT solver, making it explicit when to learn specialized instances of a quantifier. Our tactics use a similar technique: nonlinear patterns in the branches of our `match goal` constructs are our generalized equivalents of triggers: they perform the same role of guiding the prover in its learning of facts derived by specializing quantified hypotheses and theorems (Dross et al. 2015; Moskal 2009).

Chapter 6

Case studies and experimental results

Our experimental results can be divided into two broad categories. We first present a collection of microbenchmarks, small handwritten Fiat programs that exercise the main capabilities of our framework in simple to moderately complex ways and serve as a good demonstration of its capabilities. We then discuss two larger-scale experiments, in which we start from realistic high-level specifications instead of hand-written refined programs, producing efficient Facade code and in one case closed Facade modules that can be fully extracted and executed. One of these large-scale experiments focuses on mostly deterministic code but exposes many cross-language optimization opportunities; the other has more examples of nondeterminism and allows us to demonstrate final assembly code that achieves excellent performance while remaining extremely lightweight.

6.1 Microbenchmarks

Our first case study is a collection of microbenchmarks. These relatively small examples focus on demonstrating and evaluating the core approach of this thesis: extraction by synthesis. To this end, we purposefully restrict the problem to the Fiat-to-Facade part:

we do not construct full Facade modules (instead, we produce the bodies of individual Facade methods, of type `Stmt`), and we start from fully refined Fiat programs instead of high-level specifications. We furthermore assume that Bedrock implementations of list operations are available as needed.

All examples discussed in this section are collected in `MicroBenchmarksAnnotated.v` in the `Benchmarks` folder of our distribution; that file also includes a copy of the compiler's output on each example. As usual, our compilation problems are stated in terms of finding a Facade program obeying a certain `ProgOk` statement, in the form `{ inputs } ?program { outputs }`. For simplicity and conciseness, we introduce an auxiliary notation quantifying over argument variables, described below:

```
ParametricExtraction
  (* Some variables that the program is parametric on *)
  #vars      …

  (* A Fiat program *)
  #program   …

  (* A set of bindings, which are assumed to be available in the current
     scope.  These bindings give access at the Facade level to #vars. *)
  #arguments …

  (* A collection mapping names to external functions *)
  #env       …
```

Examples of microbenchmarks of varying complexities, along with the corresponding extracted programs, are shown below (Figures 6.1, 6.2, 6.3). Our collection include programs manipulating machine words and lists thereof, combining arithmetic operations and tests, conditionals, folds, maps, and nondeterministic selection of arbitrary values. In total, we present 27 small programs, all handled by the same compilation tactic; compiling each of these programs takes at most a few seconds.

The following subsections describe interesting aspects of these microbenchmarks: the

Input:

```
ParametricExtraction
  #vars      x y
  #program   ret (x ⊕ (y ⊖ x)) (* ⊖ is subtraction on machine words. *)
  #arguments ⟦ `"x" ↦ x ⟧ :: ⟦ `"y" ↦ y ⟧ :: Nil
  #env       Microbenchmarks_Env.
```

Output:

```
r ≜ y - x
out ≜ x + r
```

Figure 6.1: A Fiat program performing simple arithmetic manipulations.

Input:

```
ParametricExtraction
  #vars      seq
  #program   ret (revmap (λ w ⇒ w ⊗ 2) seq)
  #arguments ⟦ `"seq" ↦ seq ⟧ :: Nil
  #env       Microbenchmarks_Env.
```

Output:

```
out ≜ List[W].nil()
test ≜ List[W].empty?(seq)
While (test = $0)
    head ≜ List[W].pop(seq)
    r ≜ 2
    head' ≜ head * r
    call List[W].push(out, head')
    test ≜ List[W].empty?(seq)
EndWhile
call List[W].delete(seq)
```

Figure 6.2: A Fiat program mapping over a sequence of machine words.

Input:

```
ParametricExtraction
  #vars       seq
  #program    (threshold ← Any;
               ret (fold_left
                      (λ acc w ⇒
                         if threshold ≺ w then acc else w :: acc)
                      seq nil))
  #arguments  ⟦ `"seq" ↦ seq ⟧ :: Nil
  #env        Microbenchmarks_Env.
```

Output:

```
random ≜ std.rand()
out ≜ List[W].nil()
test ≜ List[W].empty?(seq)
While (test = $0)
    head ≜ List[W].pop(seq)
    test₀ ≜ random < head
    If $1 == test₀ Then
      __
    Else
      call List[W].push(out, head)
    EndIf
    test ≜ List[W].empty?(seq)
EndWhile
call List[W].delete(seq)
```

Figure 6.3: A Fiat program combining arithmetic, nondeterministic constructs, and looping.

architecture of the corresponding domain-specific compiler, support for a nondeterministic choice operator picking an arbitrary machine word, a case study in extending the compiler to handle a new pattern through rewrites, and a concrete example of support for compiler intrinsics.

6.1.1 Architecture of the microbenchmark compiler

The main entry point of the microbenchmark compiler is the local `_compile` tactic. It works by repeatedly calling into a collection of small tactics:

- `start_compiling`, which transforms the simplified notation into a synthesis goal whose head is `ProgOk`, creating the existential variable corresponding to the program being synthesized.
- `subst`, `intros`, and `computes_to_inv`, which clean up the goal to facilitate the operation of other tactics.
- `compile_do_side_conditions`, a previously described (Section 4.3.4) collection of decision procedures for commonly occurring side goals.
- `_compile_rewrite_bind`, `_compile_chomp`, and `_compile_prepare_chomp`, which transform telescopes to simplify `ProgOk` goals (also previously described in section 4.3.2).
- `_compile_skip`, `_compile_map`, `_compile_fold`, and `_compile_if`, which handle particular Gallina forms.
- `compile_simple` and `compile_simple_inplace`, which handle Boolean tests and arithmetic operations and select the right lemma based on the availability in the context of arguments of the operator being compiled.

- `_compile_early_hook` and `_compile_late_hook`, two aliases of `fail` that clients may later rebind using Ltac's dynamic binding semantics to alter the behavior of the compiler.

- `_compile_mutation`, as well as `_compile_constructor` and `_compile_destructor`, which handle certain function calls, allocations of new objects, and deallocations of objects present in the context—as long as corresponding external functions are available in the environment. These tactics are useful for illustration and experimentation purposes, though in most real-life cases writing custom mutation, allocation, and deallocation logic allows more flexibility in the specification of the Bedrock-level operations. More precisely, these tactics only work if one of the available external Bedrock functions is axiomatically specified in terms of the corresponding Gallina type and according to a very specific template. In most real-life cases, however, constructors, destructors, and mutators have more complex signatures, and their specifications tend to include special corner cases—thus requiring custom, individual call rules.

6.1.2 Adding support for an additional nondeterministic operation

This section details the steps needed to teach the microbenchmark compiler about a new nondeterministic choice.

Introducing the new primitive

We start by adding a new easily recognizable primitive, `Any`, defined in terms of `Pick` and simply describing an unknown value in the program (note that the definition of `Any` doesn't say anything about the distribution of values that it should produce; refining it away into `ret 0` would be a valid choice).

```
Definition Any ≜ { x: word 32 | ⊤ }
```

Recognizing and compiling the new primitive

If we launch the `_compile` tacic on a goal containing `Any`, the compilation process gets stuck. For example, if we start with

```
ParametricExtraction
  #program (r₁ ← Any;
            r₂ ← Any;
            ret (r₁ ⊕ r₂))
  …
```

Then `_compile` gets stuck trying to synthesize the following program, as no part of the `_compile` tactic knows how to handle `Any`.

```
{ Nil }
   ?p
{ ⟦ Any as a ⟧ :: ⟦ Any as a₀ ⟧ :: ⟦ `"out" ↦ a ⊕ a₀ ⟧ :: Nil } ∪ { ∅ } // env
```

Teaching it about this new construct is relatively easy. In this particular case, we choose to defer the nondeterministic choice to a low-level Bedrock function, which we specify thus:

```
Definition FAny {av} : AxiomaticSpec av.
Proof.
  refine {|
      PreCond ≜ λ args ⇒ args = [];
      PostCond ≜ λ args ret ⇒ args = [] ∧ ∃ w, ret = SCA av w
    |}; crush_types.
Defined.
```

We can then prove the following call rule for `Any`; this matches the basic shape of the goal that the `_compile` tactic got stuck on, though it will require a slight generalization to be applicable:

```
Lemma Any_characterization:
  ∀ x : W, Any ⇝ x.
Proof. constructor. Qed.
Hint Immediate Any_characterization : call_helpers_db.

Lemma CompileCallAny:
  ∀ {av} (env : t (FuncSpec av)),
  ∀ fpointer tenv,
    MapsTo fPointer (Axiomatic FAny) env →
    ∀ var ext,
      var ∉ ext →
      NotInTelescope var tenv →
      { tenv }
        var ≜ fPointer()
      { ⟦ `var ⇝ Any ⟧ :: tenv } ∪ { ext } // env.
Proof.
  repeat (SameValues_Facade_t_step || facade_cleanup_call); facade_eauto.
Qed.
```

At this point, all that remains to be done is to add a tactic that calls this lemma at the appropriate time: simply applying it won't do, since it assumes that adding the Any binding is the only change that is being made by the program being synthesized (for a discussion of alternative ways to phrase call rules, see section 4.2.5):

```
Ltac _compile_any ≜
  match_ProgOk
    ltac:(λ prog pre post ext env ⇒
            match constr:((pre, post)) with
            | (_, Cons _ Any _) ⇒
              call_tactic_after_moving_head_binding_to_separate_goal
              ltac:(apply CompileCallAny)
            end).
```

This tactic recognizes a postcondition whose head starts with Any, applies a transitivity lemma using the call_tactic_after_moving_head_binding_to_separate_goal primitive, and finally applies the CompileCallAny lemma. With this tactic added to the compiler the compilation process can complete, and we get the following Facade program (assuming that the FAny function is available under the name std.rand in the environment of imported functions):

```
any ≜ std.rand()
any₀ ≜ std.rand()
out ≜ any + any₀
```

6.1.3 Case study in extending the compiler

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

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

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

```
{ ⟦ `"seq" ↦ seq ⟧ :: Nil }
   ?p
{ ⟦ `"out" ↦ map (λ w ⇒ w ⊗ 2) seq ⟧ :: Nil } ∪ { ∅ } // env
```

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

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

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

```
seq' ≜ List[W].nil()
test ≜ List[W].empty?(seq)
While (test = $0)
    head ≜ List[W].pop(seq)
    call List[W].push(seq', head)
    test ≜ List[W].empty?(seq)
EndWhile
call List[W].delete(seq)
out ≜ List[W].nil()
test ≜ List[W].empty?(seq')
While (test = $0)
    head ≜ List[W].pop(seq')
    r ≜ 2
    head' ≜ head * r
    call List[W].push(out, head')
    test ≜ List[W].empty?(seq')
EndWhile
call List[W].delete(seq')
```

6.1.4 Hooks and intrinsics

One last type of interesting extensions to the compilation chain is intrinsics. Many operations can be computed using Gallina alone but admit much faster low-level implementations. As an example, here is a Gallina program that checks whether a nibble (a machine word assumed to be < 16) is a power of two (the call to `simpl` guarantees that `Inb` is fully expanded):

```
Definition nibble_power_of_two_p (w: W) ≜
  Eval simpl in bool2w (Inb w (map Word.natToWord [1; 2; 4; 8])).
```

One use of such a function might be as follows:

```
ParametricExtraction
  #vars      x
  #program   ret (nibble_power_of_two_p (x ⊕ 1))
  #arguments ⟦ `"x" ↦ x ⟧ :: Nil
  #env       env.
```

We can tell _compile to handle this function by unfolding it, by adding unfold nibble_power_of_two_p to _compile_early_hook:

```
Ltac _compile_early_hook ::= progress unfold nibble_power_of_two_p.
```

The resulting code, however, is rather inefficient:

```
r ≜ $1
l ≜ x + r
r ≜ $1
test ≜ l == r
If $1 == test Then
    out ≜ $1
Else
   r ≜ $1
   l ≜ x + r
   r ≜ $2
   test₀ ≜ l == r
   If $1 == test₀ Then
      out ≜ $1
   Else
      r ≜ $1
      l ≜ x + r
      r ≜ $4
      test₁ ≜ l == r
      If $1 == test₁ Then
         out ≜ $1
      Else
```

```
         r ≜ $1
         l ≜ x + r
         r ≜ $8
         test₂ ≜ l == r
         If $1 == test₂ Then
            out ≜ $1
         Else
            out ≜ $0
         EndIf
      EndIf
   EndIf
EndIf
```

Instead, we could choose to explicitly rely on an external implementation of the `nibble_power_of_two_p` function; it would then be the responsibility of the linker to ensure that such an implementation is available. For this, we extend the environment of imports `env` with an extra function (`FacadeImplementationWW` lifts a `W → W` function into an axiomatic specification):

```
("intrinsics", "nibble_pow2") ↦
Axiomatic (FacadeImplementationWW nibble_power_of_two_p)
```

The end result is much more succinct, and can be linked against a low-level implementation of `nibble_power_of_two_p`.

```
r ≜ $1
arg ≜ x + r
out ≜ intrinsics.nibble_pow₂(arg)
```

6.2 Binary encoders

As our first realistic benchmark, we extended the microbenchmark compiler to handle a new problem domain: that of serializing simple packets for emission over a wire. This entails taking as input a description of the serialization protocol — that is, the precise description of the bit-level layout of the data structure after serialization, as a function taking the original data types to a list of bytes — and producing a reasonably efficient program writing the corresponding bytes to an output stream. With small generalizations, this could handle commonly used packet formats such as TCP, UDP, or IPv4 packets.

Concretely, an input specification may look like the following:

```
Record MixedRecord ≜
  { f₁ : byte;
    f₂ : BoundedNat 8;
    f₃ : BoundedList (BoundedNat 8) (pow₂ 8) }.

Definition MixedRecord_encode (r: MixedRecord) ≜
       encodeW WO~0~0~1~0~1~0~1~0
  Then encodeW r.(f₁)
  Then EncodeBoundedNat r.(f₂)
  Then EncodeBoundedNat (BoundedListLength r.(f₃))
  Then EncodeBoundedList EncodeBoundedNat r.(f₃).
```

The expected output is similar to the following (the `w~0` and `w~1` notations stand for single-bit extensions of the machine word `w`):

```
tmp ≜ $128
out ≜ ByteString.new(tmp)
arg ≜ (wzero 24)~0~0~1~0~1~0~1~0
call ByteString.push(out, arg)
arg ≜ f₁
call ByteString.push(out, arg)
arg ≜ f₂
call ByteString.push(out, arg)
arg ≜ List[W].length(f₃)
call ByteString.push(out, arg)
test ≜ List[W].empty?(f₃)
While (test = $0)
    head ≜ List[W].pop(f₃)
    arg ≜ head
    call ByteString.push(out, arg)
    test ≜ List[W].empty?(f₃)
EndWhile
```

```
call List[W].delete(f₃)
```

6.2.1 The binary encoders domain-specific language

The high-level specifications given in this experiment are expressed using a Fiat DSL for binary encoders, elaborated by members of the PLV group at MIT CSAIL. This DSL offers various encoding combinators, each of which takes an output stream (in our case, a list of bytes), a cache (void in our case), and a value to encode, and each of which returns a new copy of the output stream and cache, extended to include the newly encoded value. The sources thus have a very regular structure, made even more apparent by using notations to express the composition operator. This is convenient for compilation and for automatically deriving decoders, though we do not leverage that part of the binary encoders framework in this project.

Out of the box, Fiat's binary encoders library include combinators for encoding lists and numbers into streams of Booleans or bytes, as well as a number of more complex structures that we do not consider here. These combinators are chained together through the `compose` function, previously denoted as `Then`:

```
Definition compose E B
           (transformer : Transformer B)
           (encode₁  : E → Comp (B * E))
           (encode₂  : E → Comp (B * E)) ≜
  (λ e₀ ⇒
    `(p, e₁) ← encode₁ e₀;
    `(q, e₂) ← encode₂ e₁;
    ret (transform p q, e₂)).
```

Note that all of these definitions use Bedrock's dependent `word` type; we often abbreviate `word 32` as `W` or $\mathtt{W}_{32}$:

```
Inductive word : ℕ → Set ≜
```

```
| W0 : word 0
| WS : 𝔹 → ∀ n, word n → word (S n).
```

6.2.2 Could binary encoders be extracted to OCaml?

Though the original encoding specifications may be nondeterministic, we require the refined programs to be deterministic. Since encoding generally does not require complex data structures, nondeterminism does not provide significant advantages in this particular case.

After refinement, the original encoding programs are therefore deterministic: in theory, this means that we could use the traditional OCaml extractor to obtain binaries from these sources. Doing so, however, yields disappointing results: the extracted code is multiple thousands of lines long and prohibitively slow. This is due to multiple factors:

- The source programs manipulate the dependently type "machine word" data type. Extracting this type naively leads to bad performance and code bloat: in Fiat, operations on this type are specified by roundtripping in and out of unbounded integers, and bit manipulations are done using recursive functions over the individual bits. Instead, when the number of bits in a given word matches one of the native integer types available in the target language, one could hope to extract directly to simple integer types. Unfortunately, the OCaml extraction engine is not able to extract the word type nonuniformly, so that e.g. `word 32` would be extracted to `int32`, and `word 8` to `char`.

- The source programs produce output by explicitly reconstructing bytes from the data stored inside an input packet. For numbers, for example, this means using repeated division to extract individual bits and store them inside a value of type `word`. For strings, this means iterating over the characters of the string and then iterating over the bits of the representation of each character. This makes it easy to ensure that the source programs have proper semantics, but it is a performance nightmare.

- The source programs produce sequences of bytes, with no particular focus on efficiency: they append to the end of a list using direct concatenation (concretely, to push a single byte out, they use `bytes ++ [new_byte]`). Again, this makes the sources easy to read as a description of the serialization protocol, but it yields horrible performance if extracted directly. Instead, we would like to produce an uninterrupted chunk of machine memory as the output of the encoding process: if the source program produces `[0x68; 0x65; 0x79; 0x20; 0x3A; 0x29]`, we really wish that output to be represented in memory as a contiguous array containing exactly these bytes. Thus mirroring the exact semantics of the source program into OCaml is undesirable, both for performance and for code complexity (we would have to write additional, unverified code to convert the list of bytes produced by our compiled programs into an OCaml array).

- The source programs manipulate explicit records containing the fields of the packet being serialized; this means that if we want to expose our code to, say, a C library, that library will have to construct a C `struct` for us, which we will then read values from. It would arguably be better, performance-wise, to receive each of these fields as parameters to the encoding function (additionally, it would be best to be easily interoperable with such C code, instead of having to massage the C data to be able to pass it to our extracted encoders). Thus, it would be best for users to have a say on the low-level representation of the data manipulated by these binary encoders.

Fortunately, we can fix each of these issues:

- We don't need to extract the `word` data type uniformly; instead, we can define separate instances of the `FacadeWrapper` type class, thereby encoding both bytes and 32-bit words into Facade scalars. Of course, using 32-bit scalars to represent 8-bit words is only correct as long as we don't do arithmetic on these words: the semantics of e.g. addition on 8-bit words are not the same as those on 32-bit words.

- If we choose the right machine representation for the data structures of the source programs, and if we choose the right machine-level representation of integers and strings, we can ensure that their in-memory layout will match the encoding used by the source program. Concretely, if the source program manipulates integers smaller than 2^8, we can choose as a wrapper a function that injects such numbers into 8-bit Bedrock words. With this, the code that takes such a number, extracts each bit using repeated division, and concatenates these bits to a Bedrock word can be compiled into code that copies the input number: since it is already represented at the machine level in exactly the way we wish to encode it, the encoding operation is a no-op. Similarly, if we ensure that strings are represented as contiguous arrays of chars, then encoding a string becomes essentially a memory copy from one array of chars to another.

- We can recognize the pattern that consists in appending a single byte to a list of bytes, and choose as a wrapper for list of bytes a function that injects that list of bytes into a contiguous stack of bytes. Then appending maps to the push operation of these stacks, providing a constant-time implementation of this otherwise linear-time operation.

- We are free to specify the form of the input to our encoders and decoders, and thus we can phrase the extraction problem so as to enforce that the program start with one argument variable per field in the original structure. Additionally, by carefully selecting the machine representation of each field, we can ensure easy interoperability with C code.

6.2.3 Extracting binary encoders to Facade

Set-up

Given a record type to encode and a function expressed in the binary encoders DSL serializing instances of this type, users are required to define or specify the following elements:

- Low-level representations (in terms of Bedrock types) for each of the values stored in the record. As a simple example, given the following record, users may choose to store the first number as a 32-bit machine word and the list of numbers as a vector of 32-bit machine words:

```
Record ExampleRecord ≜
{ f₁ : BoundedNat 8;
  f₂ : list (BoundedNat 8) }.
```

- A function to construct an initial memory state from the record. This function expresses the low-level way in which arguments should be passed to the compiled program fragment: in which order and under which names. Concretely, on an instance `r` of the simple record type above, this function may construct the following telescope as a precondition:

```
⟦ `"f1" ↦ r.(f₁) ⟧ :: ⟦ `"f2" ↦ r.(f₂) ⟧ :: Nil
```

Data structures

Compiled binary encoders leverage one main low-level data structure, a string buffer implemented as a flat array of bytes, which in addition to reading from particular offsets supports mutable appends to the end of the string. For efficiency, this structure is pre-allocated to a certain capacity; thus, pushes are only valid if we can produce a proof *at compile time* that there will always be enough space left in the string.

String buffers are specified to expose the following methods:

- `ByteString.New()`: allocate a string.
- `ByteString.Delete(string)`: deallocate a string.
- `ByteString.Copy(string)`: create a fresh copy of a string.
- `ByteString.Push(string, char)`: grow the string by one character.
- `ByteString.Get(string, offset)`: read one character from the string.
- `ByteString.Put(string, offset, value)`: replace one character of the string.

Bedrock's low-level implementation of string buffers uses contiguous, pre-allocated chunks of memory, with an additional length field.

Wrappers

For convenience, we provide the following wrappers in addition to previously defined ones:

- Injections from bounded natural numbers (`{x | x < pow₂ n}` where $n \leq 32$) into Facade scalars (32-bit words). These injections are straightforward but rely on uniqueness of comparison proofs to guarantee injectivity.
- Injections of lists of bounded length into regular lists.
- Injections into lists of wrapped values, given an injection into the base type. Concretely, this means that given a type `A` and an injection from `A'` into `A`, we can inject

`list A'` into `list A`. This was not necessary for microbenchmarks because they assumed for convenience that Bedrock lists were parametric in the types of their elements (to get a concrete implementation, we would thus have needed to duplicate the Bedrock specifications to cover all types used in practice). In this benchmark, on the other hand, we are interested in linking against concrete implementations; thus we specify just one type of Bedrock lists (containing stack-allocated values) and wrap other scalar values as appropriate.

Architecture of the compiler

The binary encoders compiler leverages most previously written compilation tactics and adds a number of custom ones. They handle the following parts of the compilation process:

- **Initialization**: when starting to compile, we must allocate a string buffer and an auxiliary variable to store the cache.
- **Decomposition**: encoders are written as sequences of `compose` (denoted with the `Then` operator) operations, which tactics break down into individual encoding steps.
- **Specialized rewritings**: Certain encoding steps can be rewritten to make the structure of an encoding operation more apparent. For example, the Fiat-level function the function that encodes an integer does so by (inefficiently) appending each of its individual bits to an output stream. Due to the in-memory representation chosen for integers, however, we can rewrite this operation into a simple Facade `Push` writing the byte itself into the output stream.

After a simple initialization phase in which a string buffer and a variable holding the cache are allocated, the binary encoders compilation logic keeps the goal in roughly the same shape throughout the compilation process: the head of the postcondition is an anonymous cons cell, containing a pair of a byte stream and a cache value (in all of our examples, the cache is a dummy unit value). Further bindings in the tail of the telescope are named and correspond to the actual list of bytes that we are interested in producing, plus intermediate values such as the cache (the `snd` of the head value).

The list of bytes, which we inject into a Bedrock-level string buffer, is extracted from the `fst` of the head value of the postcondition. For bookkeeping purposes the binary encoders framework uses a structure slightly more complex than a simple list of bytes (it also keeps a single byte on the side to accumulate values when encoding fields of length less than a full byte), but we store in the telescope only the list-of-bytes part of that structure. Thus, for append operations to be valid, we must ensure that the list of bytes will always be properly byte-aligned.

Finally, at the end of the postcondition telescope, we usually leave an existential variable; this allows intermediate encoding operations to freely introduce temporary values.

This particular structure is maintained as long as the encoding operation in the head cell of the postcondition telescope is complex, in the sense that it contains multiple encoding steps (i.e. as long as the head cell contains a value formed using `compose`). Once the head has been reduced to encoding a single value, such as one of the fields of the input record or a value computed therefrom, a rewriting lemma is used to make the result of this encoding apparent (as a monadic `ret` of a pair of a stream and a cache). It then becomes possible to simplify the head binding of the telescope, thus making it clear how the stream and the cache should be transformed.

At this point, the call rule for the `Push` method of the string buffer can usually be applied. Two specialized decision procedures are used to prove that the call is legitimate (that is, that the stream will always be properly byte-aligned at this point of the target program and that the underlying buffer has not reached full capacity; these tactics are described below).

Compilation rules, lemmas, and specialized tactics

Existing compilation rules cover most of the needs of the binary encoders. Additional rules that we need to prove include relatively straightforward call rules for string buffers (allocation and appending, the latter of which is presented in figure 6.4), as well as a few fairly simple additional rules to read and copy bounded integers, which had not cropped up in previous examples. Finally, we need one strong induction rule for loop compilation and a few specialized decision procedures; their complexity makes them worth discussing in more detail.

```
Lemma CompileCallPush capacity :
∀ (v_arg  v_stream  : string) (stream : list byte)
  (tenv tenv' tenv'': Telescope ADTValue)
  (n : byte) ext env p_Arg  p_Next  fPush,
  List.length stream + 1 ≤ wordToNat capacity →
  PreconditionSet tenv' ext [v_stream  ; v_arg ] →
  MapsTo fPush (Axiomatic BytesADTSpec.Push) env →
  { ⟦ `v_stream  ↦ stream ⟧ :: tenv }
    p_Arg
  { ⟦ `v_stream  ↦ stream ⟧ :: ⟦ `v_arg  ↦ n ⟧ :: tenv' } ∪ { ext } // env →
  { ⟦ `v_stream  ↦ stream ++ [n] ⟧ :: tenv' }
    p_Next
  { ⟦ `v_stream  ↦ stream ++ [n] ⟧ :: tenv'' } ∪ { ext } // env →
  { ⟦ `v_stream  ↦ stream ⟧ :: tenv }
    p_Arg
    call fPush(v_stream  , v_arg )
    p_Next
  { ⟦ `v_stream  ↦ stream ++ [n] ⟧ :: tenv'' } ∪ { ext } // env.
```

Figure 6.4: A call rule for the Facade ByteString.Push method.

A stronger loop compilation lemma In previous examples (Section 4.3.3) we had been able to compile maps and folds over lists of values as while loops. The basic template was that, if we could produce a loop body that accumulated an arbitrary single value of the right type into an arbitrary accumulator of the right type, then we were able to compile a left fold.

In the binary encoders case, however, this rule is not strong enough. Indeed, in the general case in which the accumulation function is (roughly) `λ stream elem ⇒ stream ++ [encode elem]`, we cannot provide a function that given an arbitrary buffer corresponding to a list `stream` and an element `elem` produces a buffer corresponding to the list `stream ++ [encode elem]`. This due to our choice of low-level representation for streams: we may only translate an append (++) into a mutable `Push` if the capacity of the underlying string buffer has not been reached; otherwise, the preconditions of the call rule for the string buffer's `Push` method are not met. Thus, we must be able to prove that the length of `stream` is strictly less than the capacity of the underlying buffer every time we invoke the call rule for `_.Push`; and of course we cannot do this for an arbitrary `stream`.

And yet, if we know beforehand that the capacity of the buffer is enough to accommodate all the (encoded) elements of an input list, we should be able to compile the encoding of the full list into a loop that encodes each element one by one. The key lies in strengthening the lemma so that it requires providing a loop body that, given an accumulator obtained from folding over a prefix of the input list and an element corresponding to the first value following that suffix, constructs the updated accumulator. With this more precise characterization of the input list, it becomes much more straightforward to prove that the preconditions of the call rule for the `push` method are met, since we can explicitly bound the length of the string buffer after *n* iterations of the loop as the length of the original string buffer (before the loop) plus *n* (assuming each element is encoded as a single byte).

Specialized tactics for binary encoders As mentioned previously, transforming a pure append to the end of a stream of bytes into a `_.Push` of a single byte in a mutable string buffer requires proving that the stream is properly byte-aligned and that its length is below its capacity. For this, we devise two tactics, which leverage the explicit form of the stream to bound its length (`_compile_decide_write8_side_condition`) and ensure that it is byte-aligned (`_compile_decide_padding_zero`). The core idea, in both cases, is that concatenating two byte-aligned byte streams sums their lengths and preserves their byte alignment. All that then remains is to prove that individual encoding operations preserve byte-alignment and to bound the lengths of the results of individual encoding operations.

6.3 SQL queries

Our last experiment is another application of our compilation techniques, to a new DSL. This time, however, we evaluate the entire pipeline: we start from a restricted subset of the SQL-like DSL of Delaware et al. (2015), tweak the deductive synthesis automation to produce nondeterministic programs encoding calls to external data structures as nondeterministic choices, and assemble a domain-specific compiler able to handle these programs efficiently. We also add the required support code to support extracting full Facade modules, which in turns allows us to leverage the Facade compiler to push the process all the way down to Bedrock. Finally, we evaluate the resulting pipeline by modeling and extracting a simplified process scheduler and benchmarking the resulting binaries[1] against various database engines.

6.3.1 Motivating example

For this benchmark, we focus on using Fiat to implement the core parts of a process scheduler. We describe the operations that we want it to support in high-level terms, using a

[1]linked against an assembly implementation of a Bag data structure written by Adam Chlipala

variant of a specification language that many programmers are familiar with: SQL. We argue that a process scheduler, the component of an operating system that manages the collection of processes by creating new process records, filtering processes by status, etc. is a good example of a system that has a clear and simple specification, amenable to automatic code generation, and yet is usually hand-written in a low-level language: although database engines provide good abstractions and solid performance for data-manipulating code, the abstraction costs that they introduce—both in terms of performance and in terms of memory footprint of their runtime systems—is too high to consider including one in an operating system kernel. This example is closely inspired by Hawkins et al. (2011).

Even the programs produced by the original release of Fiat, with its derivations producing efficient, verified OCaml code, were still unsuitable for this task: they were dependent on the OCaml runtime, and its garbage collector was a significant source of performance fluctuations in our original testing.

Our Fiat-to-Facade compiler removes the dependency on a garbage collector and compiles all the way down to lightweight assembly. In addition to being verified, the resulting code is fast, lightweight, and calls into hand-verified assembly implementations of the data structures that it uses to manage memory. Taken together, the pipeline is the first instance of a verified, extensible compilation system from high level specifications to assembly, supporting arbitrary delegation to separately verified data structures.

6.3.2 Source language

Though the original SQL-like DSL presented by Delaware et al. is rather rich, we restrict ourselves to single-table queries: they are sufficient to demonstrate the main abilities of our framework. Indeed, the task of disentangling complex queries into simpler functional code manipulating complex data structures through function calls is mostly that of the refinement procedure that transforms Fiat specifications into Fiat programs. The Fiat-to-Facade domain-specific compiler, in other words, is only remotely aware of the complexity of the original queries: the code that it processes has a rather different shape from the original specifications and consists mostly of relatively simple combinations of calls to external data structures.

6.3.3 High-level specifications

The concrete specifications that we refine and then compile are shown in figure 6.5. They are expressed in an SQL-like DSL offering combinators for insertion (`Insert`), enumeration (`For`), and filtering (`Where`). These notations desugar into uses of `Pick`, as explained in Delaware et al. (2015).

Each process has three attributes: a unique process ID, a state (either `SLEEPING` or `RUNNING`), and CPU time (indicating how long the process has been running for). Each of these attributes are machine words. For efficiency, the data structures that we use to store these processes should be indexed on process ids and on states: this would speed up the retrieval of processes by state and by id.

6.3.4 Refined programs

The derivations of the original Fiat paper produced deterministic programs interweaving list manipulations with access to key-value stores implemented using AVL trees. Optimization opportunities included choosing the right indexing strategy to speed up certain accesses and reordering list computations—the latter being made possible by the absence of `ORDER BY` clauses, permitting to produce results in any order.

To facilitate its exploration of indexing strategies, the original Fiat paper further introduced a `Bag` data type, specified in terms of basic multiset operations—essentially ad-

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

    (** Insert a new process, failing if ‘new_pid’ already exists. *)
    Def Spawn (r : rep) (new_pid cpu state : Word) : rep * 𝔹 ≜
      Insert (<pid :: new_pid,
               state :: state,
               cpu :: cpu>)
        into r.Processes,

    (** Find processes by ‘state’ and return their PIDs. *)
    Def Enumerate (r : rep) (state : State) : rep * list W32 ≜
      procs ← For (p in r.Processes)
              Where (p.state = state)
              Return (p.pid);
      ret (r, procs),

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

Figure 6.5: Original specifications of the process scheduler, from Delaware et al. (2015).

dition, deletion, retrieval, and enumeration. Standard-library data structures, including AVL trees, were then proven to conform to this specification, allowing one to phrase refinement theorems in terms of general bags and plug in nested AVL trees during concrete refinement runs.

Since then, the bag interface was rephrased, generalized, and lifted into a proper Fiat ADT: its methods are now Fiat computations, each producing a new copy of the ADT's internal state in addition to the method's results. To run our last experiment, we made sure that the refinement logic left uses of the Bag ADT unrefined; this allows refined programs to be expressed in terms of manipulations of a bag instance, without committing to a specific implementation of such a bag (instead, bag operations remain expressed as operations on collections of values). The rest of the optimization scripts, and in particular aspects of choosing proper indexing strategies and optimizing list computations, were unchanged.

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

(** Enumerate *)
a ← bfind table ($state, _, _);
ret (fst a, revmap (λ x ⇒ GetAttribute x 0) (snd a))

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

Figure 6.6: Functional code derived by Fiat from the process scheduler's specifications. The logic that derives these implementations from Fiat specifications is a contribution of Benjamin Delaware.

figure 6.6 shows the results of refining each of the Process Scheduler's three method specifications. These refined programs use methods of the bag ADT: `bfind` performs an efficient lookup, either by process state (first level of indexing) or by process ID (second level of indexing), and `binsert` adds a record to an existing collection.

6.3.5 Compiling and extracting query-structures code

These refined methods are almost enough to start extracting to Facade. Before doing so, however, we need to do the following:

- **Enrich Facade's ADT type to represent database tuples and indexed collections of these tuples**; this permits us to specify operations on these basic types which concrete, low-level implementations will have to satisfy. These new structures were hinted at in section 4.2.5.

- **Decide on low-level representations of input and output values**. This fundamentally comes down to the user's choices and preferences, since it amounts to choosing how to represent each of the types manipulated by the final program in terms of newly added and previously existing types available at the Bedrock level.

- **Extend the guarantees obtained by extracting refined programs to the original specifications**; this allows us to merge proofs of correctness obtained through refinement and compilation into a single proof which, once connected to the correctness proof of the Facade compiler, guarantees that the final binaries obey the original high-level specifications. This is a one-time proof, which applies uniformly to any refinement and extraction, uniformly in the choices of low-level data representations, contributed by Benjamin Delaware.

- **Construct pre- and postconditions to drive the extraction of each method**, by translating Fiat method signatures into telescopes. These specifications are to be exposed to low-level clients of the compiled Fiat ADT and are phrased in terms of low-level machine types. The corresponding Coq derivation is an elegant bit of dependently typed programming authored by Benjamin Delaware (`ADT2CompileUnit.v`), in which Fiat signatures are used to construct axiomatic specifications phrased in terms of `ProgOk` and of the various user-selected wrappers into machine types; these are the specifications that are fed to the extraction engine. This lifting of Fiat specifications into Bedrock specifications is unverified: it is hard to think of exactly what its specification should be, beyond its actual implementation. This is not much of an issue, however, since the output is simple and readily inspected, and most importantly since callers ultimately depend on the generated axiomatic specifications of the extracted programs.

The following paragraphs detail some of these points. Figures 6.7, 6.8, and 6.9 show the result of extracting the scheduler's methods.

Data structures

To support bag operations efficiently, we added a number of new ADTs in Bedrock and wrote the corresponding specifications:

Tuples of machine words Tuples are fixed-sized random-access arrays, currently implemented in Bedrock by allocating and reading from contiguous sections of machine memory. The specifications of machine-words tuples are obtained by specializing the specifications of a more general type of tuples. They support the following operations:

- `Tuple[W].new(size)`: allocate a tuple; the values of its fields are undetermined.
- `Tuple[W].delete(tuple)`: deallocate a tuple.

```
snd ≜ Bag₂[Tuple[W]].findSecond(rep, arg)
test ≜ List[Tuple[W]].empty?(snd)
If $1 == test Then
  v₁ ≜ arg
  v₂ ≜ arg₁
  v₃ ≜ arg₀
  o₁ ≜ $0
  o₂ ≜ $1
  o₃ ≜ $2
  vlen ≜ $3
  tup ≜ Tuple[W].new(vlen)
  call Tuple[W].set(tup, o₁ , v₁ )
  call Tuple[W].set(tup, o₂ , v₂ )
  call Tuple[W].set(tup, o₃ , v₃ )
  call Bag₂[Tuple[W]].insert(rep, tup)
  tmp ≜ $0
  test₀ ≜ List[Tuple[W]].empty?(snd)
  While (test₀ = $0)
      head ≜ List[Tuple[W]].pop(snd)
      size ≜ $3
      call Tuple[W].delete(head, size)
      test₀ ≜ List[Tuple[W]].empty?(snd)
  EndWhile
  call List[Tuple[W]].delete(snd)
  ret ≜ $1
Else
  tmp ≜ $0
  test₀ ≜ List[Tuple[W]].empty?(snd)
  While (test₀ = $0)
      head ≜ List[Tuple[W]].pop(snd)
      size ≜ $3
      call Tuple[W].delete(head, size)
      test₀ ≜ List[Tuple[W]].empty?(snd)
  EndWhile
  call List[Tuple[W]].delete(snd)
  ret ≜ $0
EndIf
```

Figure 6.7: Imperative code extracted from the scheduler's Spawn method.

```
snd ≜ Bag₂[Tuple[W]].findFirst(rep, arg)
ret ≜ List[W].new()
test ≜ List[Tuple[W]].empty?(snd)
While (test = $0)
    head ≜ List[Tuple[W]].pop(snd)
    pos ≜ $0
    head' ≜ Tuple[W].get(head, pos)
    size ≜ $3
    call Tuple[W].delete(head, size)
    call List[W].push(ret, head')
    test ≜ List[Tuple[W]].empty?(snd)
EndWhile
call List[Tuple[W]].delete(snd);
```

Figure 6.8: Imperative code extracted from the scheduler's `Enumerate` method.

```
snd ≜ Bag₂[Tuple[W]].findSecond(rep, arg)
ret ≜ List[W].new()
test ≜ List[Tuple[W]].empty?(snd)
While (test = $0)
    head ≜ List[Tuple[W]].pop(snd)
    pos ≜ $2
    head' ≜ Tuple[W].get(head, pos)
    size ≜ $3
    call Tuple[W].delete(head, size)
    call List[W].push(ret, head')
    test ≜ List[Tuple[W]].empty?(snd)
EndWhile
call List[Tuple[W]].delete(snd);
```

Figure 6.9: Imperative code extracted from the scheduler's `GetCPUTime` method.

- `Tuple[W].copy(tuple)`: copy a tuple.
- `Tuple[W].get(tuple, offset)`: access an element at an arbitrary offset in the tuple.

In our low-level implementation (written by Adam Chlipala), tuples are represented as contiguous arrays of machine words.

Lists of tuples This ADT is very similar to previously mentioned lists of words; it is obtained by specialization of a more general type of homogeneous lists of ADT values. One notable difference from lists of words is that, due to the single ownership rule that Facade enforces, inserting a tuple into a list clears the argument variable that contained the tuple. The methods of this ADT are the following:

- `List[Tuple[W]].new()`: allocate a list of tuples.
- `List[Tuple[W]].delete(list)`: deallocate an empty list of tuples.
- `List[Tuple[W]].empty?(list)`: test for emptiness.
- `List[Tuple[W]].push(list, value)`: add an element to the front of the list.
- `List[Tuple[W]].pop(list)`: remove an element from the front of the list.
- `List[Tuple[W]].length(list)`: query the length of the list as a machine word.
- `List[Tuple[W]].rev(list)`: mutably reverse the list.

In the low-level implementation that we link against, lists are implemented as singly linked lists.

Collections of tuples, either unindexed or indexed by one or two distinct fields This family of types describe collections of tuples indexed by certain fields; Bedrock has variants for zero, one, and two indexed fields. The implicit contract is that the underlying implementation should be optimized for speed of access to lists of tuples matching a particular key on a particular field. These collections ("bags") support the following operations:

- `Bag[Tuples].New(tlength, keys…)`: allocate a bag, keyed on `keys`, ready to hold tuples of size `tlength`.
- `Bag[Tuples].Insert(bag, tuple)`: insert a tuple into a bag
- `Bag[Tuples].Enumerate(bag)`: create a list of (copies of) all tuples contained in the bag; no guarantees are made on the order of the elements
- `Bag[Tuples].Find(bag, key)`,
  `Bag[Tuples].FindFirst(bag, key)`,
  `Bag[Tuples].FindSecond(bag, key)`,
  `Bag[Tuples].FindBoth(bag, key₁, key₂)`: enumerate tuples matching a key or (optionally, for doubly indexed collections) multiple keys. The match is an equality test between the original key and the corresponding field of the tuples.

Bedrock's low-level implementation uses nested binary trees indexed on each keyed field; find operations on the first key are implemented by enumerating all the elements of the corresponding nested tree, whereas find operations on the second tree are implemented by merging results of querying each nested tree (a process usually called "skip-scanning").

Wrappers

We then enriched the collection of Facade wrappers to store lists of Fiat tuples as lists of Bedrock tuples and bags of Fiat tuples as variously indexed Bedrock bags. Injecting lists of Fiat tuples into lists of Bedrock tuples is relatively straightforward, but matching up Fiat bags with Bedrock bags is more arduous, in particular due to the injectivity requirement.

For practical reasons, Fiat and Bedrock do not share their definitions of multisets, though they use similar ones. In both cases, bags are specified in terms of `Ensembles` (Coq's name for sets represented as functions from elements of a type to propositions describing whether an element is present in the set) of *indexed* elements—pairs of numbers (represented as machine words) and actual elements, with an accompanying well-formedness condition ensuring that no two pairs share the same index. This is a convenient way to encode multisets, ensuring that the same element can be included multiple times, as long as it is given distinct indices.

Proving the injectivity of the wrapper representing Fiat bags as Bedrock bags required lifting the injectivity of the representation of tuples as lists of words, connecting each of the individual definitions. The details of this development are in `EnsemblesOfTuplesAndListW.v`.

Architecture of the compiler

Each of the methods of the process scheduler is lifted into a pair of a pre- and a postcondition, to form a `ProgOk` statement that the Fiat-to-Facade compiler can use. The Bedrock-level types of each argument, including the ADT's internal state as well as return values, are user-provided; the lifting function decomposes the ADT's internal state (a tuple of tables) into individual variables. The domain-specific compiler for query structures is assembled, and reuses basic compilation blocks, in much the same ways as the binary encoders compiler. It is, however, comparatively simpler, owing to the smaller number of cross-language optimizations being applied:

- An **initialization phase**, `__compile_start_compiling`, sets the problem up; it is comparatively more complex than in the binary encoders case, because we are now extracting all methods of an ADT, instead of a single one. This tactic breaks down the initial record, creates an existential variable for each method's body, discharges the proof of refinement between each method's operational and axiomatic specifications, and creates a goal for each remaining proof obligation in addition to the compilation goals themselves.

- A **compilation phase** combines basic logic inherited from the microbenchmarks with custom tactics translating manipulations of bags into corresponding operations on Bedrock bags.

Call rules

Just like in the binary-encoders case, our process scheduler development benefits greatly from existing compilation rules. Its heavy use of custom data structures, however, requires additional call rules to be proven; these rules connect the semantics of these newly added data structures to the function calls that source Fiat programs encode as nondeterministic choices. Some of these rules, just like in the binary encoders case, have complex side conditions; these are discharged using specialized automation. In the following paragraphs, we highlight two difficulties specific to query-structures-related call rules.

Large proof contexts slow down automation Although their proofs are mostly automated, the complexity of the query-structures call rules makes parts of the automation slow: it is not uncommon to accumulate hundred of hypotheses in the largest proofs and to wait for multiple seconds before a proof completes. In these cases we start by proving a minimal rule expressed in terms of relatively simple states and covering only the call to the Bedrock method. We then prove a specialized version whose pre-and post-conditions match the goals that appear in practice, with set-up and tear-down operations surrounding the function call itself. This neatly separates reasoning about the semantics of Facade calls and reasoning about intermediate states and sequencing of multiple Facade statements.

Interface mismatches require complex reasoning Though the nondeterministic choices left by the refinement procedure often match up nicely with the low-level specifications of the corresponding methods, some of them exhibit small discrepancies. The discrepancies are made more difficult to handle by the injections between Fiat and Bedrock types: such inconsistencies are to be expected (there is no particular reason to expect the low-level libraries to expose interfaces precisely matching our high-level specifications), but working around them is sometimes complex.

Bedrock bags, for example, describe the `Bag.find` operation as returning a list whose elements match those obtained by filtering the original multiset with a given predicate. That Bedrock multiset, however, contains Bedrock's encoding of tuples, namely lists of machine words. On the Fiat side, on the other hand, the `bfind` function filters a set of proper Fiat tuples. To prove that their behaviors align, we assume that we start in a machine state containing a multiset of Bedrock tuples related to a multiset of Fiat tuples. From there, we must show that calling the Bedrock `Bag.find` method produces a machine state containing a list of Bedrock tuples related to a list of Fiat tuples that could be obtained by enumerating the original Fiat multiset after filtering it.

The semantics of `Bag.find`, however, only tell us that the resulting list contains the same elements as the filtered Bedrock multiset, containing Bedrock tuples; to show that this aligns with the filtering operation on Fiat tuples, we must exhibit a filtered list of Fiat tuples, whose injection into a Bedrock list of tuples equals the result of the Bedrock function calls and whose elements are the same as those of the filtered Fiat multiset (this amounts to showing that converting lists of Fiat tuples to lists of Bedrock tuples commutes with filtering multisets). The easiest way to complete such a proof is to exhibit an inverse of the conversion to lists of words, which in turn requires significant work to prove that the various filtering and conversion operations preserve tuple lengths.

Specialized tactics used in the query-structures compiler

The last part of the query-structures compiler is its collection of custom decision tactics, used to discharge side goals inherited through call rules from the preconditions of low-level bag structures used to implement indexed collections of Fiat tuples. In particular, most bag operations require proving that the bag is functional; that is, that no two elements share the same index. Additionally, inserting into a Bedrock bag requires providing a tight upper bound on the indices of its elements. In both cases, a relatively simple tactic (`__compile_discharge_bag_side_conditions`) is sufficient to discharge each of these goals as they appear during compilation.

Producing a Facade module and compiling to Bedrock

Once the programs are extracted, all that remains to be done is to package them in a closed Facade module. As mentioned earlier (Section 2.2.1), such a module contains code, along with proof obligations that are generally easy to discharge directly. The only non-trivial proof is the one that connects the module's exported axiomatic specifications (in our case phrased in terms of the `ProgOk` statement through pre- and postconditions based on telescopes) to the operational semantics of its individual methods. Fortunately, each instance of this property follows from a general theorem about `ProgOk`, connecting this class of proofs to the definition of `ProgOk` itself: the operational behavior of a Facade program that satisfies a `ProgOk` condition matches the axiomatic specifications that naturally follow from its pre- and postconditions.

With this Facade module in hand, we can invoke the Facade to Cito compiler, followed by the Cito to Bedrock compiler, then link against existing verified implementations of each of the data structures that we use, and finally extract x86 libraries. A small amount of hand-written x86 driver code is added to make it easy to call each of the methods from C. The refinement from specification to code in Fiat takes less than fifteen minutes; extracting then takes another few minutes, and compiling and extracting Bedrock assemblies then takes about a day (there are no fundamental reasons for this slowness).

6.3.6 Evaluation

Benchmark design and experimental setup

We started by running small tests to ensure that our code was indeed returning the right results; these helped as a sanity check that everything had been properly put together (for example, it helped us ensure that our small amount of driver code had been correctly written).

Then, to evaluate the efficiency of our code and submit it to light integration testing, we wrote a C driver that created a large pool of process objects and used our ADT's methods to manage them. More precisely, the program would:

- Create a pool of processes, with about 10 in `RUNNING` state and all others ins `SLEEPING` state, and insert them all in a new process collection.
- Enumerate running processes (repeatedly).
- Get the CPU time of a random process (repeatedly).

We replicated this very same experiment with three major database management systems: PostgreSQL, MySQL, and SQLite. We wrote queries equivalent to the ones that we had written in Fiat's dialect of SQL and replicated the process of creating and retrieving process records. We set up a single index on the pair of columns `(status, pid)`, mirroring the organization of the data in the Fiat implementation. In all cases we disabled commits during tests, and in SQLite's case we used the in-memory mode to provide a fairer comparison — though of course caveats apply. Our solution is guaranteed to be correct with regards to its original specifications but does not guarantee anything about recovering from hardware crashes, for example; the database systems that we benchmarked against, on the other hand, are unverified but provide stronger informal guarantees.

The next section details our results.

Experimental results

Figure 6.10 shows the running times of each query in each system (MySQL was omitted, because its performance was worse than that of both SQLite and PostgreSQL). Our initial results showed both SQLite and PostgreSQL at a strong disadvantage in the `GetCPUTime` case (`/CPU` curves on the diagram): upon investigation, we realized that all three database systems were using an inefficient query plan, doing a linear search over the entire table instead of performing a skip-scan and thereby getting an unexpectedly linear time complexity for that query. To force these systems to use the appropriate query plan, we had to add a clause to the query:

```
-- Original query
   SELECT cputime FROM Processes WHERE pid = ?
-- Updated query
   SELECT cputime FROM Processes WHERE status IN (0, 1) AND pid = ?
```

With this change, all three engine were able to take advantage of the index for the `GetCPUTime` query (`/CPU'` curves on the diagram). On the `GetCPUTime` query our code is an order of magnitude faster than SQLite and two orders of magnitude faster then PostgreSQL; on the `Enum` query our performance is comparable to SQLite and more than an order of magnitude better than PostgreSQL.

These tests serve as a reassuring sanity check: they show that our pipeline is able to erase most of the runtime cost of starting from high-level specifications instead of hand-crafting low-level programs, and they confirm that we achieve the right asymptotic complexity, comparing favorably to systems taking input specifications similar to ours — with the added benefit that we provide proofs of correctness for the resulting binaries and do not require a heavy runtime system.

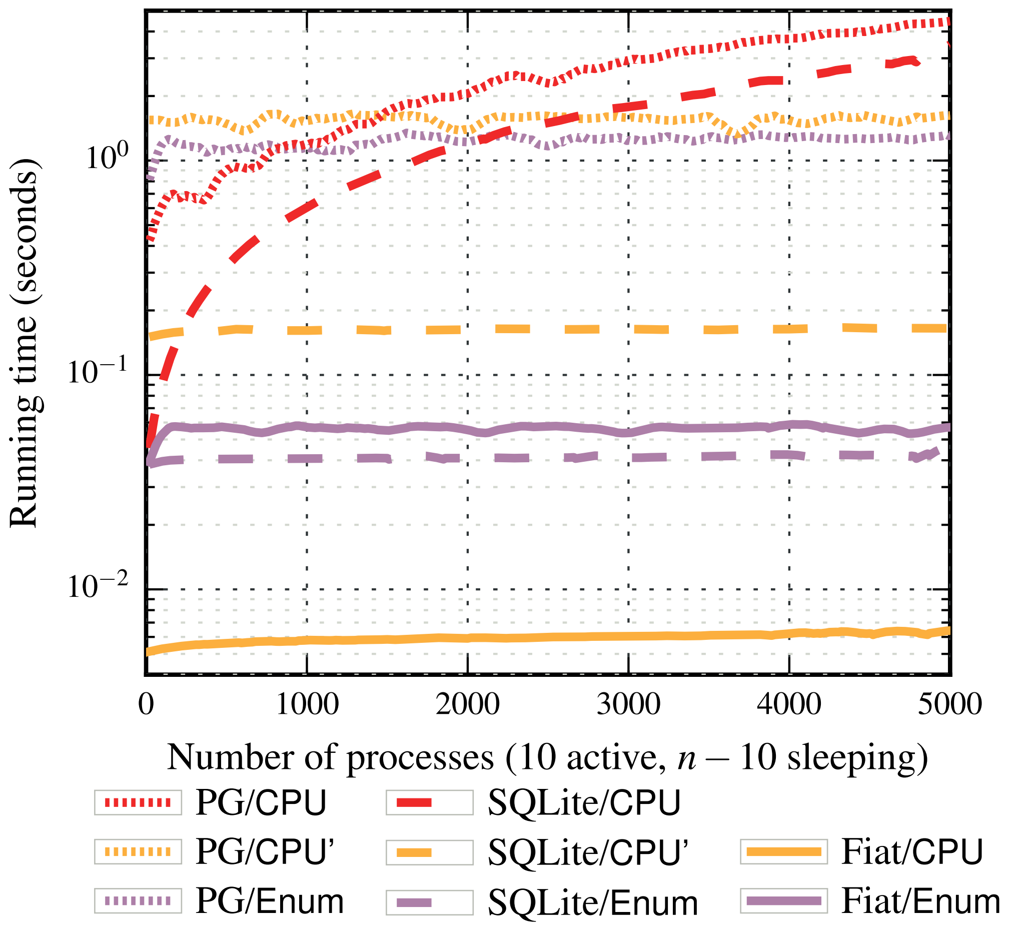

Figure 6.10: Experimental results (absolute performance and scalability) obtained on our process-scheduler benchmark. We measure the running time of 20 000 `Enumerate` and 10 000 `GetCPUTime` queries after inserting increasingly large numbers of processes. We maintain the invariant that the number of processes in the `RUNNING` state is about 10. "PG": PostgreSQL; "CPU": Unmodified `GetCPUTime` query; "CPU'": `GetCPUTime` query modified to tip PostgreSQL and SQLite about proper index use; "Enum": `Enumerate` query.

Chapter 7

Other approaches and related work

Many recently completed and currently ongoing research projects have focused on topics closely related to that of this thesis. Among them, four projects are most closely related to our work. In all cases, our project puts a stronger emphasis on extensibility, at the cost of completeness; this generally allows us to consider a broader class of optimizations and crucially supports our vision of compilers as open collections of small compilation routines.

7.1 Imperative/HOL

In the context of Isabelle/HOL, Lammich (2015) has developed a related tool that uses separation-logic based pre- and postconditions. This tool targets the Imperative/HOL monadic language and differs from ours both in spirit and in implementation: where Lammich uses separation logic, we rely instead on sequent-calculus style derivations with preconditions expressed over Facade's simplified view of memory. Similarly, where Lammich introduces explicit linearity checks in the course of the derivation, we rely instead on Facade's lightweight post-hoc syntactic checks to establish linearity. Unlike Lammich's tool, we support nondeterministic source programs, and we allow linking with externally implemented data structures. Finally, both tools have different application targets and application domains: while Lammich applies his work to classic textbook algorithms, we focus on relatively simple code leveraging complex externally implemented data structures, coming closer, arguably, to the type of software that programmers usually write.

7.2 CakeML extraction

Building on the CakeML verified compiler (Kumar et al. 2014), Myreen and Owens (2012) have built a proof-producing pipeline from HOL light into CakeML. Thanks to the very close correspondence between the source and target language, the translation is relatively straightforward: when the source program is expressed in the state monad it can be translated into effectful code, and when it is phrased in a more functional style it can leverage CakeML's functional features. Our pipeline, on the other hand, targets more restricted but further remote languages. This gives us many ways to phrase complex optimizations that are otherwise hard to express: we get precise control over memory allocation and deallocation and freedom to introduce calls to externally implemented, axiomatically specified data structures, delaying linking until a later stage.

7.3 Cogent

In a recent development, O'Connor et al. (2016) have presented Cogent, a linearly typed language whose semantics closely mirror those of Facade. This language does allow calls to external data structures, but it puts significantly stronger constraints on the source programs: in a sense, it becomes the programmer's task to phrase their programs so as to make the translation into a lower-level language straightforward. In that sense, Cogent is much closer to a traditional verified compiler; our pipeline, in contrast, sacrifices completeness for extensibility, preferring an unconstrained source language (Gallina in the nondeterminism monad) and relying on compiler extensions to handle new patterns. This gives us the flexibility to produce programs differing significantly from their sources, without committing a specific interface of external calls, nor particular data structures, in the source program.

7.4 Reflection and denotation

A different approach to the extraction problem is convincingly being explored by members of the Œuf team at the University of Washington (Mullen et al. 2016), drawing inspiration from reflective proof techniques. Instead of interleaving compilation to a lower-level language and extraction from Gallina, this project uses an unverified Ltac program to reify Gallina's syntax into a deeply embedded inductive type; the output of each such extraction is validated by denoting the resulting AST back into Gallina syntax. The rest of the pipeline follows a traditional verified compilation approach, starting from the resulting deeply embedded Gallina terms and producing low-level assembly through a verified C compiler (Leroy 2006). This approach is straightforward and relatively lightweight and works very nicely in this project's context, focusing on simple, deterministic programs, with no specific focus on cross-language optimization opportunities. It would, however, not be a very good fit for our goals. In particular, it places constraints on the source language (the denotation approach is more complex to implement if the source language is e.g. extended to allow fixpoints) and, more importantly, does not straightforwardly support calls to externally implemented functions or uses of nondeterministic choices, nor significant changes in data-structure representation as part of the compilation process. Additionally, it restricts optimizations to those that can be expressed as total functions over deeply embedded ASTs; our approach, on the other hand, does not place such constraints on optimizations: since they are expressed as Gallina lemmas and applied during extraction, rules are free to depend on generally undecidable preconditions—it is enough for these preconditions to be decidable in every specific instance in which the rule is used.

In addition to this most closely related work, this thesis connects to a number of related domains: program extraction in proof assistants, translation validation, compiler verification, extensible compilation, and program synthesis.

7.5 Program extraction

Many proof assistants, starting with NuPrl (Constable et al. 1986) and more recently Coq (The Coq development team 2015), use extraction facilities to produce executable code from shallowly embedded functional programs. Though extraction algorithms have sometimes been proved on paper, their implementations have not. They are, however, relatively complex pieces of software, for example involving complex reasoning about erasure of `Prop` subterms in the case of Coq. The resulting programs are often hard to audit due to the introduction of unsafe primitives of the target language (`Obj.magic`), used to bypass the limitations of OCaml's weaker type system.

7.6 Translation validation

An alternative approach, recently used in the context of verified operating-system kernel development (Sewell et al. 2013), 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 it generally falls short when trying to verify complex optimizations. One of the most fascinating uses of translation validation is the work on seL4 (Klein et al. 2009), 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 (Peleska et al. 2011) and Z3 (Moura and Björner 2008).

Many verified compilers, and in particular CompCert, rely on unverified programs to compute solutions to complex problems, such as register allocation. Instead of verifying a full register allocator, it is simpler to depend on an external solver to produce a good allocation and to employ a smaller, verified checker to guarantee the soundness of the solver's output. This approach works well for complex problems with easily checkable solutions (register allocation, for example, is equivalent to graph coloring and thus NP-complete), but it does not directly apply to our case: it is undecidable whether a given Facade program faithfully implements a Fiat source program. Hence we crucially need to produce, in addition to a Facade program, a proof of its correctness. In that sense, one could see our strategy as a validating approach where in addition to a solution to the compilation problem (which on its own would be impossible to check) the compiler produces a witness of correctness that can be validated by Coq's type checker. The need to produce this witness then makes it natural to do the synthesis as a dialogue with a proof assistant.

7.7 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 (Leroy 2006) or CakeML

(Kumar et al. 2014) (in the former, correctness guarantees indeed only extend to linking modules compiled with the same version of the compiler): CompCert does not allow users to link their programs against manually implemented and verified performance-critical software libraries. More recent work (Stewart et al. 2015) 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. Finally, a recently launched project, CertiCoq (Appel et al. 2016), is attempting to build a verified compiler for Gallina starting with an unverified reflection step to obtain a Gallina AST before using on a multi-pass verified compiler to obtain assembly code.

7.8 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 (Tobin-Hochstadt et al. 2011), do not focus on verification. Other efforts, such as the Rhodium system (Lerner et al. 2005), 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 (Tatlock and Lerner 2010), 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 seen as a generalization of this approach: our entire compiler is written as a collection of transformations, each of which gets us closer to a fully constructed Facade program.

7.9 Program synthesis

Manna and Waldinger (1980) pioneered the view of program synthesis as a deductive verification task in 1980, and indeed our work views extraction as a proof-producing synthesis process, in which we progressively construct the value of an undetermined existential variable to produce a full Facade program matching a source Fiat program. We put strong requirements on the shapes of our pre- and postconditions so that we can use their syntactic structure to drive synthesis, thereby restricting the search space and facilitating automation. This idea of strongly constraining the search space is found in work by Solar-Lezama (2009) on the *Sketch* language, as well as Alur et al. (2013). In both cases, the core idea is to supplement the specification from which a program is synthesized by an implementation template, which leaves parts empty for the synthesizer to generate. In contrast to our work, though, these programs use a general-purpose solver to explore the space of candidate programs; our compiler, on the other hand, applies a user-extensible collection of rules to recognize and compile patterns.

Conclusion

The synthesis-based extraction technique that this thesis presents offers a convenient and lightweight approach to develop domain-specific compilers from shallowly embedded languages to deeply embedded imperative languages, reducing the trusted base of verified programs to the kernel of a proof assistant and safely exposing to users a whole new class of optimization opportunities. Beyond phrasing extraction as a synthesis problem leveraging an extensible collection of verified compilation rules to drive a dialogue with a proof assistant, this thesis details our implementation and demonstrates the applicability of our ideas to real-world problems. In particular, we push the guarantees offered by nondeterministic programs obtained by refinement from high-level specifications all the way down to a verification-aware assembly language, thereby closing the last gap in the first mechanically certified translation pipeline from declarative specifications to assembly-language libraries, supporting user-guided optimizations and abstraction over arbitrary ADT implementations.

Future work

The results that we obtained in the course of this thesis are a rich ground for future work: we could leverage Facade's support for recursive functions and add support for recursion, and we could apply this to new domains such as automatically derived parsers. It would also be interesting to explore alternate formulations of our basic `ProgOk` statement: indeed, preliminary work by members of the FSCQ (Chen et al. 2015) project, as they started to apply the ideas presented in this thesis, suggests that a more compact encoding based on isolating a single nondeterministic computation (and expressing postconditions as a function from values of this computation to unordered maps) could be used as well.

Acknowledgments

Many people contributed to this thesis. I thank my advisor Adam Chlipala for his guidance, advice, and dedication; my colleague and at times mentor Ben Delaware for his help and suggestions; my fellow students Jason Gross and Peng Wang for their technical contributions, and Ben Sherman for careful proofreading and valuable suggestions; and my former professors Stéphane Graham-Lengrand, Éric Goubault, and David Monniaux for introducing me to some of the topics discussed in this thesis and helping me decide to pursue a PhD. On a more personal note, I thank my family for their support, understanding, encouragements, and for many more reasons than feasible to list here; my friends Letitia Li for help with the online version of this document, Thomas Bourgeat for proofreading early drafts, and Max Dunitz for sharing his thesis-writing experience; and finally Reyu Sakakibara for English language guidance and cupcakes.

Bibliography

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa (2013). "Syntax-guided synthesis". *Proceedings of the 2013 Formal Methods in Computer-Aided Design Symposium - FMCAD '13*. DOI: 10.1109/fmcad.2013.6679385.

Andrew W. Appel, Greg Morrisett, Randy Pollack, Matthieu Sozeau, Olivier Savary Belanger, Zoe Paraskevopoulou, and Abhishek Anand (2016). "CertiCoq: Principled Optimizing Compilation of Dependently Typed Programs". URL: https://www.cs.princeton.edu/~appel/certicoq/.

David Aspinall (2000). "Proof General: A Generic Tool for Proof Development". *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*. Volume 1785. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pages 38–43. DOI: 10.1007/3-540-46419-0_3.

Samuel Boutin (1997). "Using Reflection to Build Efficient and Certified Decision Procedures". *Proceedings of the 3rd International Symposium on Theoretical Aspects of Computer Software - TACS '97*. Springer Science + Business Media, pages 515–529. DOI: 10.1007/bfb0014565.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich (2015). "Using Crash Hoare Logic For Certifying the FSCQ File System". *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*, pages 18–37. DOI: 10.1145/2815400.2815402.

Adam Chlipala (2013). "The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier". *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming - ICFP '13*. DOI: 10.1145/2500365.2500592.

Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith (1986). *Implementing Mathematics with the Nuprl Proof Development System*. NJ: Prentice-Hall. DOI: 10.2307/2274489.

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala (2015). "Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant". *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15*. Association for Computing Machinery (ACM), pages 689–700. DOI: 10.1145/2676726.2677006.

Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich (2015). "Adding Decision Procedures to SMT Solvers Using Axioms with Triggers". *Journal of Automated Reasoning* 56.4, pages 387–457. DOI: 10.1007/s10817-015-9352-2.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo (2015). "Deep Specifications and Certified Abstraction Layers". *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15*. Association for Computing Machinery (ACM), pages 595–608. DOI: 10.1145/2676726.2676975.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv (2011). "Data Representation Synthesis". *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '11*. Association for Computing Machinery (ACM), pages 38–49. DOI: 10.1145/1993498.1993504.

Gerwin Klein, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, and et al. (2009). "seL4: Formal Verification of an OS Kernel". *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*. Association for Computing Machinery (ACM), pages 207–220. DOI: 10.1145/1629575.1629596.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens (2014). "CakeML: A Verified Implementation of ML". *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '14*. Association for Computing Machinery (ACM). DOI: 10.1145/2535838.2535841.

Peter Lammich (2015). "Refinement to Imperative/HOL". *Interactive Theorem Proving*. Volume 9236. Lecture Notes in Computer Science. Springer International Publishing, pages 253–269. DOI: 10.1007/978-3-319-22102-1_17.

Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers (2005). "Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules". *Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages - POPL '05*. DOI: 10.1145/1040305.1040335.

Xavier Leroy (2006). "Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant". *Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '06*. Association for Computing Machinery (ACM), pages 42–54. DOI: 10.1145/1111037.1111042.

Pierre Letouzey (2003). “A New Extraction for Coq”. *Selected Papers from the 2nd International Workshop on Types for Proofs and Programs - TYPES ’02*. Springer Science + Business Media, pages 200–219. DOI: 10.1007/3-540-39185-1_12.

Pierre Letouzey (2004). “Programmation fonctionnelle certifiée – L’extraction de programmes dans l’assistant Coq”. Thèse de Doctorat. Université Paris Sud. URL: https://tel.archives-ouvertes.fr/tel-00150912.

Zohar Manna and Richard J. Waldinger (1980). “A Deductive Approach to Program Synthesis”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1, pages 90–121. DOI: 10.1145/357084.357090.

Michał Moskal (2009). “Programming with Triggers”. *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories - SMT ’09*. Association for Computing Machinery (ACM), pages 20–29. DOI: 10.1145/1670412.1670416.

Leonardo de Moura and Nikolaj Björner (2008). “Z3: An Efficient SMT Solver”. *Tools and Algorithms for the Construction and Analysis of Systems*. Volume 4963. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pages 337–340. DOI: 10.1007/978-3-540-78800-3_24.

Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman (2016). “Œuf: Verified Coq Extraction in Coq”. URL: http://oeuf.uwplse.org/.

Magnus O. Myreen and Scott Owens (2012). “Proof-Producing Synthesis of ML from Higher-Order Logic”. *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming - ICFP ’12*. Association for Computing Machinery (ACM). DOI: 10.1145/2364527.2364545.

Liam O’Connor, Christine Rizkallah, Zilin Chen, Sidney Amani, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Alex Hixon, Gabriele Keller, Toby C. Murray, and Gerwin Klein (2016). “COGENT: Certified Compilation for a Functional Systems Language”. *CoRR* abs/1601.05520. URL: http://arxiv.org/abs/1601.05520.

Jan Peleska, Elena Vorobev, and Florian Lapschies (2011). “Automated Test Case Generation with SMT-Solving and Abstract Interpretation”. *NASA Formal Methods*. Volume 6617. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pages 298–312. DOI: 10.1007/978-3-642-20398-5_22.

Clément Pit-Claudel and Pierre Courtieu (2016). “Company-Coq: Taking Proof General one step closer to a real IDE”. *CoqPL’16: The Second International Workshop on Coq for PL*. DOI: 10.5281/zenodo.44331.

Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein (2013). “Translation Validation for a Verified OS Kernel”. *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI ’13*. Association for Computing Machinery (ACM), pages 471–482. DOI: 10.1145/2491956.2462183.

Armando Solar-Lezama (2009). “The Sketching Approach to Program Synthesis”. *Proceedings of the 7th Asian Symposium on Programming Languages and Systems - APLAS ’09*. Springer Berlin Heidelberg, pages 4–13. DOI: 10.1007/978-3-642-10672-9_3.

Matthieu Sozeau (2009). “A New Look at Generalized Rewriting in Type Theory”. *Journal of Formalized Reasoning* 2.1, pages 41–62. DOI: 10.6092/issn.1972-5787/1574.

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel (2015). “Compositional CompCert”. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL ’15*. Association for Computing Machinery (ACM). DOI: 10.1145/2676726.2676985.

Thomas Streicher (1993). “Investigations into Intensional Type Theory”. Habilitation thesis. Ludwig-Maximilians-Universität München. URL: http://www.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf.

Zachary Tatlock and Sorin Lerner (2010). “Bringing Extensibility to Verified Compilers”. *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI ’10*. Association for Computing Machinery (ACM), pages 111–121. DOI: 10.1145/1806596.1806611.

The Coq development team (2015). *The Coq Proof Assistant Reference Manual*. Version 8.4pl6. LogiCal Project. URL: http://coq.inria.fr.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen (2011). “Languages as Libraries”. *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI ’11*. Association for Computing Machinery (ACM), pages 132–141. DOI: 10.1145/1993498.1993514.

Philip Wadler (1995). “Monads for functional programming”. *Lecture Notes in Computer Science*. Springer Science + Business Media, pages 24–52. DOI: 10.1007/3-540-59451-5_2.

Peng Wang (2016). *The Facade Language*. Technical report. MIT CSAIL. URL: http://people.csail.mit.edu/wangpeng/facade-tr.pdf.

Peng Wang, Santiago Cuellar, and Adam Chlipala (2014). “Compiler Verification Meets Cross-Language Linking via Data Abstraction”. *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications - OOPSLA ’14*. Association for Computing Machinery (ACM), pages 675–690. DOI: 10.1145/2660193.2660201.