Relational compilation for end-to-end verification

by

Clément Pit-Claudel

MS, Massachusetts Institute of Technology (2016)
Dipl. Ing., École Polytechnique (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

December 2021

Signature of author: ______________________________

Department of Electrical Engineering and Computer Science
December 2, 2021

Certified by: ______________________________

Adam Chlipala
Associate Professor of Computer Science
Thesis Supervisor

Accepted by: ______________________________

Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Relational compilation for end-to-end verification

by

Clément Pit-Claudel

Submitted to the Department of Electrical Engineering and Computer Science on December 2, 2021 in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Computer Science.

ABSTRACT

Purely functional programs verified using interactive theorem provers typically need to be translated to run: either by extracting them to a similar language (like Coq to OCaml) or by proving them equivalent to deeply embedded implementations (like C programs). Traditionally, the first approach was automated but produced unverified programs with average performance, and the second approach was manual but produced verified, high-performance programs.

This thesis shows how to recast program extraction as a proof-search problem to automatically derive correct-by-construction, high-performance code from shallowly embedded functional programs. First, it introduces a unifying framework, *relational compilation*, to capture and extend recent developments in program extraction, with a focus on modularity and sound extensibility. Then, it presents Rupicola, a relational compiler-construction toolkit designed to extract fast, verified, idiomatic low-level code from annotated functional models.

The originality of this approach lies in its combination of foundational proofs, extensibility, and performance, backed by an unconventional take on compiler extensions: unlike traditional compilers, Rupicola generates good code not because of clever built-in optimizations, but because it allows users to plug in domain- and sometimes program-specific extensions soundly. This thesis demonstrates the benefits of this approach through case studies and performance benchmarks that highlight how easy Rupicola makes it to create domain-specific compilers that generate code with performance comparable to that of handwritten C programs.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor of Computer Science

TODO: Compiled 2021-12-20T01:21:43, git revision 19770bf690072759df3b5c1dc9ec9b563ae7db8a

Acknowledgments

I am deeply grateful to the many people who supported me during my PhD, foremost among them my advisor, Adam Chlipala, and my family: my wife, my brother, and my parents. I feel immensely fortunate to have benefited from their support, encouragement, and advice over all these years.

I also would like to thank my colleague and mentor, Ben Delaware, who worked closely with me in the first years of my PhD; my coworker Jade Philipoom, with whom I created the first versions of Rupicola, and the other members of the Rupicola team at MIT; my friends, colleagues and classmates at MIT, from whom I kept learning throughout my PhD, especially Zelda Mariet, Sara Achour and Cecilia Testart; my close friend Thomas Bourgeat, who showed infinite patience with my coffee-drinking habits back when research was an in-person activity; my professors and committee members at MIT, especially Sam Madden, Joel Emer, Daniel Jackson, Jonathan Ragan-Kelley, and Frans Kaashoek; all the professors, friends and mentors who supported me professionally and trusted me, especially Jonathan Protzenko, Stéphane Lengrand, Éric Goubault, Rustan Leino, Zach Tatlock, and Arvind; my internship supervisors and colleagues, especially Cătălin Hrițcu and Nik Swamy; my SIGPLAN-M mentor, Patrick Lam, who helped me navigate the last year of my PhD; my wonderful colleague in organizing CAV 2021's artifact evaluation, Maria A Schett; the members of the interactive theorem proving research community, especially the members of Coq's development team; the researchers and volunteers who give their time to develop and maintain the tools that I use in my research; and all the friends who supported me and kept in touch despite my terrible email habits, especially Letitia Li, Max Dunitz, Clément Feltin, Caroline Vu Tien and Olivier Hercend.

Table of contents

1 Introduction

Vulnerabilities in critical systems fall into roughly two categories: logic mistakes (incorrect business logic) and programming mistakes (use-after-free, out-of-bounds accesses, etc.). High-level languages attempt to eliminate logic mistakes by promoting higher levels of abstraction that facilitate reasoning about program behavior, and they protect against low-level issues using static and dynamic checks and safer programming paradigms (garbage collection to rule out use-after-free errors, stream- and result-oriented APIs for out-of-bounds accesses, etc.).

At one extreme, purely functional languages offer very strong protections against low-level mistakes and readily lend themselves to mathematical reasoning. By eliminating arrays, exceptions, state, and other low-level concerns and encouraging higher-order programming, languages like Coq [58], Lean [10], Idris [5], or the pure subsets of Haskell and F* [56, 18] offer programming models much less susceptible to the low-level issues that plague the vast majority of today's critical systems.

Unfortunately, this combination of flexibility and safety comes at a significant performance cost: it is an unsolved problem to program a compiler for any of these purely functional languages that verifiably preserves all of their high-level guarantees while offering performance competitive with the usual low-level suspects, especially C.

In fact, to ground this discussion of high-level inefficiencies, let us consider the simple task of converting an ASCII string str to uppercase, maybe as part of a network program that receives a request and normalizes its contents to use them as a key in a table of records.

In a purely functional language like Gallina (part of the Coq proof assistant), this task can be implemented succinctly as follows (with strings being linked list of characters, characters an inductive type with 256 cases, and toupper a switch with one case per lowercase letter):

```
String.map Char.toupper str
```

This program accurately captures the intent of the task, but how fast does it run? When extracted to OCaml, it will pointer-chase through a linked list to traverse the original string (creating data dependencies and cache pressure), create a fresh string (costing allocations, cache misses, and an extra traversal for garbage collection), and either stack-overflow on long strings (due to a non-tail-recursive map, though there have been recent developments in that space), or traverse the string twice (doubling allocation and pointer-chasing costs), or accumulate continuations (even more allocations).

In contrast, the low-level implementation below performs a single pass, occupies constant stack space, does not allocate, is cache-friendly, can be unrolled, and is trivially vectorizable (`toupper` on ASCII chars is just a comparison and a bitmask): it assumes that the original string is never reused, represents `str` as a contiguous array of characters, and mutates it with a simple `for` loop.

```
for (int i = 0; i < len; i++)
  str[i] = toupper(str[i]);
```

It is possible to see this low-level program as a transformation of the high-level one above, but only if we broadly generalize the way we think about compiler extensions.

Rethinking compiler extensions — The traditional extension point in a compiler is a single-language rewrite rule (e.g., in GHC Haskell), and much past research has studied compiler optimizations through the lens of term-rewriting systems. In fact, many compilers are implemented as sequences of lowering passes interleaved with optimization passes that operate on single languages. Unfortunately, single-language rewrite rules are poorly suited to the kind of cross-language transformations with potentially complex side conditions that would be needed to automatically generate the C program above from the Coq one preceding it.

Expressing the translation of `String.map` into `for` loop with mutation as a rewrite rule, for example, would require us either to encode mutation and for loops explicitly in the source language (so that the transformation could be performed within Gallina), or to extend C to support higher-order functions and `folds` (so that the transformation could be performed within C), or to use a generic compiler to lower `folds` into C and subsequently eliminate all the resulting cruft (including closures and a GC) using more rewrite rules. This expressivity issue, in addition to the fact that such transformations are often conditional on relatively complex side conditions that are best solved by user-provided partial decision procedures, makes rewrite rules poorly suited to our problem.

The aim of this thesis is to make it possible to build custom compilers that support complex cross-language optimizations, allowing users to translate the functional `String.map` program above into an efficient in-place loop. Such transformations are crucial to get good performance, easy to express for specific programs or domains, but either too narrow in scope or too complex to generalize to be a good fit for a general-purpose compiler.

A different kind of compiler — To realize this vision, this thesis describes a different approach to the compilation of functional programs. Its defining characteristic is that it trades completeness for performance: in my approach, users assemble custom compilers for specific programs or collections of programs, and as a result these compilers do not need to support the full complexity of the source language. By abandoning completeness, code patterns that would be complex and costly to compile in full generality (e.g. requiring garbage collection, closures, a runtime system, etc.) can be mapped to simple low-level constructs like loops and mutations by exploiting domain- or program- specific assumptions.

This vision is realized as a compiler-construction toolkit, Rupicola, implemented in the Coq proof assistant. In Rupicola, users assemble compilers by combining individual *theorems* that connect high-level patterns in Gallina (Coq's functional programming language) to low-level code fragments in Bedrock2 (a low-level imperative language developed at MIT [14]). So, for example, a Rupicola user may prove a theorem that translates all `maps` on lists into `for` loops that mutate arrays in place. Such an implementation choice is not applicable to all uses of `maps` and lists: it works only if mutation is acceptable (i.e. if the original list is not needed anymore after the `map`), and if the original code neither adds to nor removes from the list (these operations have no direct equivalent on a fixed-length array).

Rupicola is not intended to replace all program extraction. Instead, Rupicola is restricted, out of the box, to a minimal set of constructs (essentially arithmetic, simple data structures, and some control flow), yielding a predictable and transparent compilation process. Users are expected (and enabled) to extend it as needed for each new domain, plugging in domain- or program-specific compilation hints that capture the insight that humans would normally apply when manually implementing high-level specifications in a low-level language: details of memory layout and memory management, implementation strategies for data-structure traversals, etc.

Figure 1.1 shows where Rupicola fits in a complete pipeline from high-level specifications to assembly code.

Rupicola's target audience and use cases — Rupicola works best for small, performance-critical programs, where precise control over implementation choices and optimization

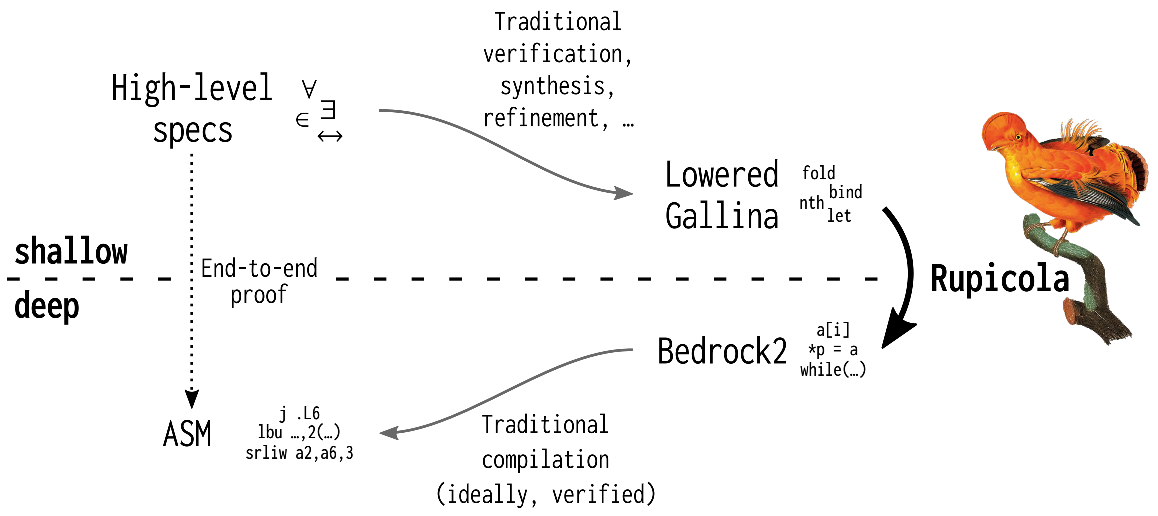

Traditional verification, synthesis, refinement, …
High-level specs
∀ ∃ ∈ ↔
Lowered Gallina
fold bind nth let
shallow
deep
End-to-end proof
Rupicola
Bedrock2
a[i] *p = a while(…)
ASM
j .L6 lbu …,2(…) srliw a2,a6,3
Traditional compilation (ideally, verified)

Figure 1.1: Where Rupicola fits in the bigger picture. Rupicola is not a universal compiler: it bridges the gap between shallowly embedded purely functional programs and deeply embedded imperative code by accepting a restricted (but extensible) input that can reliably and predictably be translated to fast low-level code. Right: Rupicola's namesake, the Guianan *cock-of-the-rock*, *Rupicola Rupicola*.

is crucial — the kind of programs that experts write directly in C, to avoid the overheads introduced by traditional functional-programming compilers. In other words, Rupicola's user are intended to know what kind of low-level code they want, and Rupicola's task is to allow them to generate that code from functional models instead of writing it directly. Rupicola's value proposition, in this use case, is a combination of automation and ease of reasoning:

- Transformations that would be repeatedly applied by hand when manually implementing a low-level program from a high-level description are instead encoded a single time as compiler extensions, and subsequently applied many times across a range of related programs. Users pay a bit more upfront (they have to encode the transformation into a Rupicola plug-in) but reap the benefits down the line.
- Reasoning about the source programs becomes much simpler: since the source programs are valid Gallina code, proofs about these programs do not have to deal with the subtleties of low-level languages like mutation, complex control flow, or memory allocation. This makes verifying code from end to end much easier, because all program-specific

reasoning and proofs happen on shallowly embedded purely functional programs, which are only then translated into efficient imperative code.

In a sense, Rupicola codifies and automates away the most unpleasant part of traditional end-to-end verification pipelines. In the traditional world, authors not willing to rely on Coq's extraction (for performance or trust reasons) will manually relate handwritten, deeply embedded low-level programs to functional models, and then they will separately relate each functional model to a high-level specification. In that world, authors must repeatedly deal with the complexities of the low-level language's semantics and with details such as when to allocate or free memory or how to relate low-level memory layouts to high-level functional models. Rupicola, in contrast, completely automates the first phase of this process, generating the deeply embedded low-level program from its functional model by leveraging user-provided hints and program annotations. In Rupicola, programmers only supply *shallowly* embedded programs written in a subset of Gallina that naturally maps to low-level constructs, and the tooling produces low-level, deeply embedded code. Unchanged is the second phase that relates these functional models to abstract specifications: that part is still the programmer's responsibility.[1] But because Rupicola's inputs are shallowly embedded, this phase is disconnected from the details of the low-level language's semantics, and the traditional reasoning patterns best supported by Coq — especially structural induction — are fully applicable.

1.1 Dissertation outline

This dissertation starts with an in-depth presentation of relational compilation, the theoretical foundation that Rupicola is built on, with special emphasis on composability and extensibility. Relational compilation is a unifying framework that I developed to capture and extend recent developments in program extraction: a collection of program-derivation techniques that soundly bridges the gap from shallowly embedded programs to deeply embedded executable code. Some of the ideas that I present in this section were previously known, but the presentation itself is new, as are many of the advanced use cases that I

[1]This rule is somewhat overstated: in simple cases, Rupicola is sufficiently extensible that high-level executable specifications can in fact directly be mapped into low-level code, so the second phase can at times be eliminated entirely; such is the case of the `String.map` example of this section.

demonstrate. By leveraging this technique, I show how to construct modular, extensible domain-specific compilers that perform advanced domain- or even program-specific transformations in a safe manner.

I then provide a real-world perspective on relational compilation, using it to derive, in the Coq proof assistant, high-performance implementations of various small yet bug-prone low-level programs. The resulting framework, Rupicola, is a compiler-construction toolkit, not a standard compiler. Because our focus is on low-level programming, we[2] made no attempt to compile *all* or even *most* functional programs. Instead, we focused on relatively small, loop-oriented programs such as those often found in binary parsers, text-manipulation libraries, cryptographic routines, system libraries, and other high-risk, high-performance code. These programs are not traditionally implemented in purely functional languages, but this thesis shows that relational compilation can in fact be used to write performance-critical programs in that style, directly within the native, pure logic of an interactive theorem prover, combining straightforward reasoning and verification with excellent performance — on par with handwritten code. In other words, Rupicola's inputs are pure and written with `map`s and `fold`s, but they compile to C code that manages its own memory, mutates its inputs, and runs at the speed of vectorizable `for` loops.

To support these claims, this dissertation then presents a collection of case studies and benchmarks. The benchmarks measure the performance of code generated using Rupicola for a variety of example programs and show that it generally matches that of handwritten code (section 5.2). Case studies quantify the effort needed to plug in new translations (section 5.1.1) and expose low-level features to source programs (section 5.1.2); demonstrate benefits from switching from reification to relational compilation on Rupicola's expression compiler (a 10× reduction in Ltac code size, section 5.1.3); show how Rupicola fits within an end-to-end pipeline by measuring the effort needed to connect high-level specifications to a functional model suitable for compilation with Rupicola (section 5.1.4); and describe third-party uses of Rupicola (section 5.1.5).

1.2 Thesis contributions

This thesis claims two main contributions:

[2] Rupicola is a joint effort, so I use "we" liberally in the rest of this document to discuss design decisions. I give a detailed account of individual contributions in section 6.5 below.

- A novel, systematic presentation of relational compilation, in Coq, with a focus on composability and extensibility;
- The description and evaluation of Rupicola, a relational compiler from Gallina, Coq's programming language, to Bedrock2 [14], a low-level imperative language. Rupicola advances the state of the art through its composable support for arbitrary monadic programs, novel treatment of loops, and output-code performance.

In other words, Rupicola demonstrates that, at least for some simple loop-oriented programs, users do not have to chose between slow but reliable high-level code and fast but bug-prone low-level programs.

While the techniques that this thesis develops are presented in the context of the Coq proof assistant, they are more largely applicable. In particular, Rupicola innovates in the way it treats loops and effects, and these innovations could carry to other systems:

- Most verification systems handle loops in a unified way: for example, the semantics of Bedrock, the language that Rupicola targets, has a single deduction rule for `while` loops. This means that reasoning about loops is often done in terms of the lowest-level loop primitive (high-level loops are mapped to a low-level iteration primitive and proofs about the loops body mention symbolic state also phrased in terms of that primitive). In Rupicola, in contrast, each type of loop is compiled a custom lemma: there are distinct lemmas for maps, folds, iteration on ranges of numbers, etc.; sometimes more than one lemma for a single type of loop. These custom loop lemmas differ in the way they encode intermediate loop states: each one of them exclusively mentions the corresponding high-level iterator (`map` and `fold`), independently of the way the loop is eventually compiled. Thanks to this, invariant inference for Rupicola loops can be entirely automated (the computation of the loop's strongest postcondition is trivial), and all reasoning about loops and loop states is done at the level of purely functional code (section 4.5.2, section 4.6.2.3).
- Control over low-level features such as memory allocation and over effects such as mutation is critical for performance, but rewriting functional programs to make uses of these features explicit (e.g. using monadic encodings) is costly. Rupicola instead introduces most effects and low-level features as part of the compilation process, using lightweight annotations that are semantically transparent — that is, they do not make reasoning about the program any more complicated, and they are not reflected in the program's type (section 4.5.1.1). As a bonus, however, most of Rupicola is parametric on a choice of

monad, so even code best expressed using monads can be mapped to low-level code with minimal effort (section 4.5.1.2).

There have been many previous efforts in this space, foremost among them developments on Imperative/HOL [23, 24], CakeML [36, 19], Œuf [34, 21], HOL compilation to Verilog [30], Fiat-to-Facade [48] (my own previous work), CertiCoq [1], and Low* [52]. Rupicola's novelty is its combination of performance, foundational proofs, and extensibility:

- All projects above except CertiCoq and Low* use relational compilation pipelines. Among these, only Fiat-to-Facade focuses on generating high-performance low-level code from functional programs, but it did not come close to achieving Rupicola's performance (other relational compilers target either garbage-collected languages or other types of languages like Verilog; LLVM/HOL for example compiles from a one-to-one shallow embedding of LLVM).
- KreMLin, Low*'s compiler, does produce code with performance matching handwritten programs, but it is not formally verified (Rupicola's output is certified by a proof of total correctness). CertiCoq has proofs (though not for the initial reification step), but it is a standard compiler forcing use of a runtime system.
- Among the above, only Fiat-to-Facade strove for straightforward user extensions, but unlike in Rupicola users were limited by linearity of the target language, and support for loops and effects was ad-hoc (a loop could mutate only one object, and the nondeterminism monad was hardcoded).

2 Prelude: Interactive theorem proving and formal verification

The software artifacts that support this thesis are built within the Coq proof assistant. Coq is a venerable piece of software: its development started in 1984[3], and has over time produced a powerful and versatile programming and verification environment.

At the core of Coq is a programming language called Gallina. It resembles traditional functional languages like SML, OCaml, F#; here, for example, is how one can define a Coq function that filters a list according to a predicate:

```
From Coq Require Import List.
Import ListNotations.

Fixpoint filter {α} (p: α → bool) (l: list α) {struct l} :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ let t' := filter p t in
             if p h then h :: t' else t'
  end.
```

A function defined in this way can be executed interactively using the `Compute` command:

```
Compute filter Nat.even [1; 2; 4; 13; 42; 139; 506].
```

```
= [2; 4; 42; 506]
: list nat
```

The same function can then be *extracted* — that is, translated — to other languages, including OCaml (this is the way most Coq programs are converted into executable binaries today):

[3]For a complete history of the development of Coq, the reader is encouraged to consult the excellent *History* chapter of Coq's reference manual [59], available online at `https://coq.github.io/doc/v8.14/refman/history.html`.

```
From Coq Require Import Extraction ExtrOcamlBasic.
Extraction filter.
```

```
(** val filter : ('a1 → bool) → 'a1 list → 'a1 list **)

let rec filter p = function
| [] → []
| h :: t → let t' = filter p t in if p h then h :: t' else t'
```

Beyond these superficial similarities, Coq diverges from traditional functional languages in multiple important ways — some theoretical and some practical. For the purposes of the discussion to follow, the following are the most relevant differences:

First, there are no effects in Coq: no mutable variables, no exceptions, no nondeterminism, no I/O and no control flow except for recursion and pattern matching. As a result, computations in Coq are very similar to mathematical computations: calling the same function twice with the same arguments always returns the same values.

Second, all programs written in Coq terminate (there are no unbounded loops, only well-founded recursion); hence Coq rejects the following incorrect program, for example:

```
Fixpoint filter {α} (p: α → bool) (l: list α) {struct l} :=
  match l with
  | [] ⇒ []
  | h :: t ⇒
    let t' := filter p l in
    if p h then h :: t' else t'
  end.
```

```
Recursive definition of filter is ill-formed.
In environment
filter : forall α : Type, (α → bool) → list α → list α
α : Type
p : α → bool
l : list α
h : α
t : list α
Recursive call to filter has principal argument equal to
"l" instead of "t".
Recursive definition is:
"fun (α : Type) (p : α → bool) (l : list α) ⇒
 match l with
 | [] ⇒ []
 | h :: _ ⇒ let t' := filter α p l in if p h then h :: t' else t'
 end".
```

As a consequence, with a few exceptions irrelevant to this discussion, reduction strategies do not matter in Coq. This is important because computation is at the core of everything in Coq, and unlike most languages Coq supports partial evaluation, reduction under binders, and reduction with holes:

```
Eval cbv in filter ?[f] [1; 2].
```

```
= if ?f 1 then 1 :: (if ?f 2 then [2] else []) else if ?f 2 then [2] else []
: list nat
```

```
Eval simpl in (fun x ⇒ filter Nat.even [1; x; 4; 13; 42]).
```

```
= fun x : nat ⇒ if Nat.even x then [x; 4; 42] else [4; 42]
: nat → list nat
```

Third, Coq has a particularly powerful type system, capable of capturing not just the simple types of OCaml, but also relations between types and values — in fact, Coq does not distinguish between types and values: the types of natural numbers and lists are defined by induction, but so are the types of logical disjunctions, existentially quantified propositions, or even equalities:

```
Print nat.
```

```
Inductive nat : Set :=  O : nat | S : nat → nat
```

```
Print list.
```

```
Inductive list (A : Type) : Type :=  nil : list A | cons : A → list A -
> list A
```

```
Print or.
```

```
Inductive or (A B : Prop) : Prop :=
    or_introl : A → A \/ B | or_intror : B → A \/ B
```

```
Print ex.
```

```
Inductive ex (A : Type) (P : A → Prop) : Prop :=
    ex_intro : forall x : A, P x → exists y, P y
```

```
Print "=".
```

```
Inductive eq (A : Type) (x : A) : A → Prop :=  eq_refl : x = x
```

This allows Coq's language, Gallina, to be used not just for writing programs, but also for stating properties and proving them. For example, we can define predicates and prove theorems about the `filter` function above. The predicate contains below returns a mathematical proposition capturing whether a value is contained in a list:

```
Fixpoint contains {α} (l: list α) (x: α) : Prop :=
  match l with
  | [] ⇒ False
  | h :: t ⇒ h = x ⋁ contains t x
  end.
```

Proofs in Coq are simply values that inhabit a certain types: just like we can say that 5 has type nat (for natural numbers), we can say that `Nat.add_comm` has type `forall n m: nat, n + m = m + n` — and really what this means is that `Nat.add_comm` is a function that, given two numbers `m` and `n`, returns a proof (a value) of type `n + m = m + n`:

```
From Coq Require Import Arith.
Check Nat.add_comm.
```

```
Nat.add_comm
     : forall n m : nat, n + m = m + n
```

Here is a theorem about the `filter` function defined above. Its statement is a type, that of a function which given a function `f`, a list `l`, a value `x`, and a proof of type `contains (filter f l) x`, returns a proof that `f x` equals `true`:

```
Lemma f_if {A B} (f: A → B) (b: bool) a a':
  f (if b then a else a') = if b then f a else f a'.
Proof. destruct b; reflexivity. Qed.
```

```
Theorem filter_sound {α} (f: α → bool):
  forall l x, contains (filter f l) x → f x = true.
```

Proofs in Coq are typically written using a meta-language that generates terms under the hood; this is called the *tactic* language. Coq is an interactive system: proofs are written step by step, and after each step the system displays the current state of the proof, a collection of open “goals”, which are secondary theorems that need to be proved to complete the main proof. Let us see a concrete example; in what follows the gray boxes indicate Coq’s output, and the text without background is user input.[4]

```
Proof.
```

[4]These boxes are automatically generated using a program called Alectryon [45]

The Proof command begins the proof. Above the bar are the *hypotheses* that hold at this point in the proof. Below the bar is the *goal*, the theorem that we are trying to prove. Since the contains predicate is defined by induction on l, we follow that structure in the proof, using the induction command; this corresponds to distinguishing two cases in the proof: empty and non-empty lists.

```
induction l.
```

```
α: Type      f: α → bool
--------------------------------------------------------------------- 1
forall x : α, contains (filter f []) x → f x = true

α: Type      f: α → bool      a: α      l: list α
IHl: forall x : α, contains (filter f l) x → f x = true
---------------------------------------------------------------------
forall x : α, contains (filter f (a :: l)) x → f x = true
```

We are now presented with two "subgoals" — two cases. In the first (1) the list l has been replaced by the empty list []; in the second the list is assumed to be non-empty, and l has been replaced by a cons of a newly introduced element a and a new list l.

The all: combinator below applies a tactic to all goals, and intros moves forall-quantified variables and premises of implications into the context as hypotheses:

```
all: intros.
```

```
α: Type      f: α → bool      x: α      H: contains (filter f []) x 2
---------------------------------------------------------------------
f x = true

α: Type      f: α → bool      a: α      l: list α
IHl: forall x : α, contains (filter f l) x → f x = true      x: α
H: contains (filter f (a :: l)) x
---------------------------------------------------------------------
f x = true
```

The - below focuses the proof on the first subgoal. In that goal, hypothesis **H**: contains (filter f []) x (2) is contradictory, as filter f [] reduces to [], and contains [] x reduces to False; and indeed, using the simpl tactic to perform evaluation.:

```
- simpl in H.
```

```
α: Type      f: α → bool      x: α      H: False
---------------------------------------------------------------------
f x = true
```

What **H**: False means is that hypothesis H has type False. In Coq False is defined as an empty inductive case, so H inhabiting the type False is inconsistent: case analysis on H completes this branch of the proof:

```
      destruct H.
```

The second goal is less simple; this time, simplification suggests two cases: `f a = true` and `f a = false`, so we can perform a case analysis on that value:

```
    - simpl in H.
```

```
α: Type     f: α → bool     a: α     l: list α
IHl: forall x : α, contains (filter f l) x → f x = true     x: α
H: contains (if f a then a :: filter f l else filter f l) x
────────────────────────────────────────────────────────────
f x = true
```

```
      destruct (f a) eqn:Hf.
```

```
α: Type     f: α → bool     a: α     l: list α
IHl: forall x : α, contains (filter f l) x → f x = true     x: α
Hf: f a = true     H: contains (a :: filter f l) x
────────────────────────────────────────────────────────────
f x = true
```

```
α: Type     f: α → bool     a: α     l: list α
IHl: forall x : α, contains (filter f l) x → f x = true     x: α
Hf: f a = false     H: contains (filter f l) x
────────────────────────────────────────────────────────────
f x = true
```

We find ourselves with two new subgoals: in the first `f a` is true (as indicated by **Hf**: `f a = true`) and `filter f (a :: l)` reduced to `a :: filter f l`; in the second `f a` is false (**Hf**: `f a = false`) and `filter f (a :: l)` reduced to `filter f l`. Further simplification will suggest one more case split, as `contains (a :: filter f l)` itself reduces to a disjunction: either `a = x` or x is in the result of `filter f l`.

```
      + simpl in H.
```

```
α: Type     f: α → bool     a: α     l: list α
IHl: forall x : α, contains (filter f l) x → f x = true     x: α
Hf: f a = true     H: a = x ∨ contains (filter f l) x
────────────────────────────────────────────────────────────
f x = true
```

```
      destruct H.
```

```
α: Type      f: α → bool      a: α      l: list α
IHl: forall x : α, contains (filter f l) x → f x = true      x: α
Hf: f a = true      H: a = x
──────────────────────────────────────────────────────────────
f x = true
```

```
α: Type      f: α → bool      a: α      l: list α
IHl: forall x : α, contains (filter f l) x → f x = true [3]      x: α
Hf: f a = true      H: contains (filter f l) x
──────────────────────────────────────────────────────────────
f x = true
```

In the first case `a = x`, and we are in the case in which `f a` is true; hence the goal holds:

```
      * rewrite H in Hf.
```

```
α: Type      f: α → bool      a: α      l: list α
IHl: forall x : α, contains (filter f l) x → f x = true      x: α
Hf: f x = true      H: a = x
──────────────────────────────────────────────────────────────
f x = true
```

```
        assumption.
```

In the second case we know by assumption that x is in `(filter f l)` (3), but also by induction that all x in `(filter f l)` satisfy f (3):

```
      * apply IHl.
```

```
α: Type      f: α → bool      a: α      l: list α
IHl: forall x : α, contains (filter f l) x → f x = true      x: α
Hf: f a = true      H: contains (filter f l) x
──────────────────────────────────────────────────────────────
contains (filter f l) x
```

```
        assumption.
```

Finally we read the case in which `f a = false`, and the induction hypothesis applies immediately:

```
    + apply IHl. assumption.
```

A satisfying `Qed` closes the proof:

```
Qed.
```

Under the hood tactic generate proof terms, and in fact it is possible to write proofs directly as plain Gallina programs. The result is seldom readable, however:

```
Fixpoint filter_complete {α} (f: α → bool) l {struct l}:
  forall x, contains l x → f x = true → contains (filter f l) x :=
  match l return (forall x, contains l x → f x = true →
                  contains (filter f l) x) with
  | [] ⇒ fun x (H: False) _ ⇒ H
  | a :: l ⇒ fun x (Hc: a = x ∨ contains l x) Hf ⇒
      match Hc with
      | or_introl Heq ⇒
        eq_rect_r
         (fun a ⇒ contains (if f a then a :: _ else _) x)
         (eq_rect_r (x := true)
            (fun b ⇒ contains (if b then x :: _ else _) x)
            (or_introl eq_refl) Hf)
         Heq
      | or_intror Hc ⇒
          if f a as b return (contains (if b then a :: _ else _) x)
          then or_intror (filter_complete f l x Hc Hf)
          else filter_complete f l x Hc Hf
      end
  end.
```

A final distinguishing characteristic of Coq is its support for advanced notations: unlike traditional languages, almost all the syntax of Coq is defined through extensions of its parser; later in this document we will use this to define special syntax for dictionaries, Hoare triples, function specifications, etc.

Coq proofs are usually written using specialized IDEs [3, 47] that support showing Coq code side by side with the corresponding proof state.

For more information on the Coq proof assistant, readers can consult any of the books and tutorials listed at `https://coq.inria.fr/documentation`. In the rest of this document chapter 3 should be accessible to readers with limited Coq experience; the following chapters assume some Coq proficiency.

3 On relational compilation

The traditional process for developing a verified compiler is to define types that model the source (S) and target (T) languages, and to write a function $f : S \to T$ that transforms an instance s of the source type into an instance $t = f(s)$ of the target type, such all behaviors of t match existing behaviors of s ("refinement") and sometimes additionally such that all behaviors of s can be achieved by t ("equivalence", or "correctness").

Naturally, *proving* correctness for such a compiler requires a formal understanding of the *semantics* of languages S and T (that is, a way to give meaning to programs $s \in S$ and $t \in T$, so that it is possible to speak of the behaviors of a program: return values, I/O, resource usage, etc.). Then the refinement criterion above translates to $\sigma_T(t) \subseteq \sigma_S(s)$ (where $\sigma_S(s)$ denotes the behaviors of s and $\sigma_T(t)$ those of t), and the correctness criterion defines a relation between source and target programs such that $t \sim s$ iff. $\sigma_T(t) = \sigma_S(s)$. With that, a compiler f is correct iff. $f(s) \sim s$ for all s.

Relational compilation is a twist on that approach: it turns out that instead of writing the compiler as a monolithic program and separately verifying it, we can break up the compiler's correctness proof into a collection of orthogonal correctness theorems, and *use these theorems* to drive a code-generating proof search process. It is a Prolog-style "compilers as relations" approach, but taken one step further to get "compilers as (constructive) decision procedures".

Instead of writing our compiler as a function $f : S \to T$, we will write the compiler as a (partial) decision procedure: an automated proof-search process for proving theorems of the form $\exists t, \sigma_T(t) = \sigma_S(s)$. In a constructive setting, any proof of that statement must exhibit a witness t, which will be the (correct) compiled version of s. (Note that the theorem does not $\forall$-quantify s, as we want to generate one distinct proof *per input program* — otherwise, with a $\forall s$ quantification, the theorem would be equivalent by skolemization to $\exists f, \forall s, \sigma_T(f(s)) = \sigma_S(s)$, which is the same as defining a single compilation function f… and precisely what we're trying to avoid.)

The two main benefits of this approach are flexibility and trustworthiness: it provides a very natural and modular way to think of compiler extensions, and it makes it possible to extract shallowly embedded programs without trusting an extraction routine (in contrast, extraction in Coq is trusted). The main cost? Completeness: a (total) function always terminates and produces a compiled output; a (partial) proof search process may loop or fail to produce an output.[5]

This is not an entirely new idea: variations on this trick have been variously referred to in the literature as *proof-producing compilation*, *certifying compilation*, and, when the source language is shallowly embedded (we will get to that a bit later), *proof-producing extraction*, *certifying extraction*, or *binary extraction*. I have not seen a systematic explanation of it yet, so here is my attempt. I like to call this style of compilation "relational compilation", and to explain it I like to start from a traditional verified compiler and progressively derive a relational compiler from it.

3.1 A step by step example

Here is a concrete pair of languages that we will use as a demonstration. Language S is a simple arithmetic-expressions language. Language T is a trivial stack machine. You can download the Coq code of this example

3.1.1 Language definitions

On the left is the Coq definition of `S`, with only three constructors: constants, negation, and addition; on the right is the definition of `T`: a program in `T` is a list of stack operations `T_Op`, which may be pushing a constant, popping the two values on the top of the stack and pushing their difference, or popping the two values on the top of the stack and pushing their sum.

[5] Of course things are not so clear-cut: a compiler may be written as a partial function that sometimes fails to compile input programs, like ill-typed programs in OCaml, programs with too-deep template recursion in C++, programs with too-long lines in Python, etc. — but the extensibility of relational compilers also tends to make them more susceptible to incompleteness, and since they're written in meta-language it is often not easy or even possible to prove their completeness.

```
Inductive S :=
| SInt z
| SOpp (s : S)
| SAdd (s1 s2 : S).
```

```
Inductive T_Op :=
| TPush z
| TPopSub
| TPopAdd.

Definition T := list T_Op.
```

3.1.2 Semantics

The semantics of these languages are easy to define using interpreters. On the left, operations on terms in S are mapped to corresponding operations on ℤ, producing an integer. On the right, stack operations are interpreted one by one, starting from a stack and producing a new stack.

```
Fixpoint σS s : Z :=
  match s with
  | SInt z      ⇒ z
  | SOpp s      ⇒ - σS s
  | SAdd s1 s2 ⇒ σS s1 + σS s2
  end.
```

```
Notation Stack := (list Z).

Definition σOp (ts: Stack) op : Stack :=
  match op, ts with
  | TPush n, ts ⇒ n :: ts
  | TPopAdd, n2::n1::ts ⇒ n1+n2 :: ts
  | TPopSub, n2::n1::ts ⇒ n1-n2 :: ts
  | _, ts ⇒ ts (* Invalid: no-op *)
  end.

Definition σT t (ts: Stack) : Stack :=
  List.fold_left σOp t ts.
```

With these definitions, program equivalence is straightforward to define (the definition is contextual, in the sense that it talks about equivalence in the context of a non-empty stack):

```
Notation "t ~ s" := (forall ts, σT t ts = σS s :: ts).
```

3.1.3 Compilation

In the simplest case, a compiler is a single recursive function; more typically, compilers are engineered as a sequence (composition) of passes, each responsible for a well-defined task: typically, either an *optimization* (within one language, intended to improve performance of the output) or *lowering* (from one intermediate language to another, intended to bring the program closer to its final form), though these boundaries are porous. Here is a simple one-step compiler for our pair of languages:

```
Fixpoint StoT s := match s with
  | SInt z     ⇒ [TPush z]
  | SOpp s     ⇒ [TPush 0] ++ StoT s ++ [TPopSub]
  | SAdd s1 s2 ⇒ StoT s1 ++ StoT s2 ++ [TPopAdd]
end.
```

The SInt case maps to a stack machine program that simply pushes the constant z on the stack; the SOpp case returns a program that first puts a 0 on the stack, then computes the value corresponding to the operand s, and finally computes the subtraction of these two using the TPopSub opcode; and the SAdd case produces a program that pushes both operans in succession before computing their sum using the TPopAdd opcode.

The Coq command Compute lets us run this compiler and confirm that it seems to operate correctly:

```
Example s7 :=
  SAdd (SAdd (SInt 3) (SInt 6))
        (SOpp (SInt 2)).
```

- Running the example program s7 directly returns 7:

  ```
  Compute σS s7.
  ```

  ```
  = 7
  : Z
  ```

- Compiling the program and then running it produces the same result (a stack with a single element, 7):

  ```
  Compute StoT s7.
  ```

  ```
  = [TPush 3; TPush 6; TPopAdd; TPush 0; TPush 2; TPopSub; TPopAdd]
  : list T_Op
  ```

  ```
  Compute σT (StoT s7) [].
  ```

  ```
  = [7]
  : Stack
  ```

3.1.4 Compiler correctness

Of course, one example is not enough to establish that the compiler above works; instead, here is a proof of its correctness, which proceeds by induction with three cases:

```
Lemma StoT_Rok : forall s, StoT s ~ s.
Proof. 4
  induction s.
  all: intros; unfold σT; simpl.
  - (* `SInt` case *)
    reflexivity.
  - (* `SOpp` case *)
    rewrite fold_left_app.
    rewrite IHs.
    reflexivity.
  - (* `SAdd` case *)
    rewrite !fold_left_app.
    rewrite IHs1, IHs2.
    reflexivity.
Qed.
```

This compiler operates in a single pass, but arguably even a small compiler like this could benefit from a multi-pass approach: for example, we might prefer to separate lowering in two phases translating all unary SOpp operations to a new binary operator SSub (SOpp x → SSub (Sconst 0) x) in a first pass, and dealing with stack operations in a second pass.

3.1.5 Compiling with relations

We will observe two things about StoT and its proof.

First, StoT, like any function, can be rewritten as a relation (any function $f : x \mapsto f(x)$ defines a relation $\sim_f$ such that $t \sim_f s$ iff. $t = f(s)$; this is sometimes called the graph of the function). Here is one natural way to rephrase StoT as a relation $\mathfrak{R}$; notice how each branch of the recursion maps to a case in the inductive definition of the relation (each constructor defines an introduction rule for $\mathfrak{R}$, which corresponds to a branch in the original recursion, and each x $\mathfrak{R}$ y premise of each case corresponds to a recursive call to the function):

```
Inductive StoT_rel : T → S → Prop :=
| StoT_RNat : forall z,
    [TPush z] ℜ SInt z
| StoT_ROpp : forall t s,
    t ℜ s →
    [TPush 0] ++ t ++ [TPopSub] ℜ SOpp s
| StoT_RAdd : forall t1 s1 t2 s2,
    t1 ℜ s1 →
    t2 ℜ s2 →
    t1 ++ t2 ++ [TPopAdd] ℜ SAdd s1 s2
where "t 'ℜ' s" := (StoT_rel t s).
```

Now, what does correctness mean for StoT? Correctness for this compilation relation is... just a subset relation:

> The relation ℜ is a *correct* compilation relation for languages S and T if its graph is a subset of the graph of $\sim$.

And that condition is sufficient: any relation that is a subset of $\sim$ defines a correct (possibly one-to-many, possibly suboptimal, but *correct*) mapping from source programs to destination programs.

And indeed, the relation above is a correct compilation relation:

```
Theorem StoT_rel_ok : forall t s,
    t ℜ s → t ~ s.
Proof. 5
  induction 1.
  all: intros; unfold σT; simpl.
  - (* `StoT_RNat` case *)
    reflexivity.
  - (* `StoT_ROpp` case *)
    rewrite fold_left_app.
    rewrite IHStoT_rel.
    reflexivity.
  - (* `StoT_RAdd` case *)
    rewrite !fold_left_app.
    rewrite IHStoT_rel1, IHStoT_rel2.
    reflexivity.
Qed.
```

Now that we have ℜ, we can use it to prove specific program equivalences: for example, we can write a proof to show specifically that the compiled version of our example program s7 matches the original s7 by applying each of the constructors of the relation ℜ one by one:

```
Goal ([TPush 3] ++ [TPush 6] ++ [TPopAdd]) ++
     ([TPush 0] ++ [TPush 2] ++ [TPopSub]) ++
     [TPopAdd] ℜ
     SAdd (SAdd (SInt 3) (SInt 6))
```

```
          (SOpp (SInt 2)).
Proof.
  apply StoT_RAdd.
```

```
────────────────────────────────────────────────────────────
[TPush 3] ++ [TPush 6] ++ [TPopAdd] ℜ SAdd (SInt 3) (SInt 6)
────────────────────────────────────────────────────────────
[TPush 0] ++ [TPush 2] ++ [TPopSub] ℜ SOpp (SInt 2)
```

```
  - apply StoT_RAdd.
```

```
────────────────────────────────────────────────────────────
[TPush 3] ℜ SInt 3
────────────────────────────────────────────────────────────
[TPush 6] ℜ SInt 6
```

```
    + apply StoT_RNat.
    + apply StoT_RNat.
  - apply StoT_ROpp.
```

```
────────────────────────────────────────────────────────────
[TPush 2] ℜ SInt 2
```

```
    + apply StoT_RNat.
Qed.
```

Now, how can we use this relation to *run* the compiler instead? By using proof search! This is standard practice in the world of logic programming. To *compile* our earlier program s7, for example, we can simply search for a program t7 such that t7 ℜ s7, which in Coq terms looks like this:

```
Example t7_rel: { t7 | t7 ℜ s7 }.
Proof.
  unfold s7; eexists.
```

```
────────────────────────────────────────────────────────────
?t7 ℜ SAdd (SAdd (SInt 3) (SInt 6)) (SOpp (SInt 2))
```

Now the goal includes an inderminate value ?t7, called an existential variable (*evar*), corresponding to the program that we are attempting to derive, and each application or a lemma *refines* that evar by plugging in a partial program:

```
  apply StoT_RAdd.
```

```
────────────────────────────────────────────────────────────
?t1 ℜ SAdd (SInt 3) (SInt 6)
```

```
────────────────────────────────────────
?t2 ℜ SOpp (SInt 2)
```

After applying the lemma StoT_RAdd, we are asked to provide two subprograms, each corresponding to one operand of the addition:

```
  - apply StoT_RAdd.
```

```
────────────────────────────────────────
?t1 ℜ SInt 3
────────────────────────────────────────
?t20 ℜ SInt 6
```

```
    + apply StoT_RNat.
    + apply StoT_RNat.
  - apply StoT_ROpp.
```

```
────────────────────────────────────────
?t ℜ SInt 2
```

```
    + apply StoT_RNat.
Defined.
```

We get the exact same program, but this time instead of *validating* a previous compilation pass, we have generated the program from scratch:

```
Compute t7_rel.
```

```
= exist [TPush 3; TPush 6; TPopAdd; TPush 0; TPush 2; TPopSub; TPopAdd]
: {t7 : T | t7 ℜ s7}
```

We can also use Coq's inspection facilities to see the proof term as it is being generated (this time the interstitial boxes show the internal proof term, not the goals):

```
(exist ?t7)
```

```
  apply StoT_RAdd.
```

```
(exist (?t1 ++ ?t2 ++ [TPopAdd]))
```

```
  - apply StoT_RAdd.
```

```
(exist ((?t1 ++ ?t20 ++ [TPopAdd]) ++ ?t2 ++ [TPopAdd]))
```

```
    + apply StoT_RNat.
```

```
(exist (([TPush 3] ++ ?t20 ++ [TPopAdd]) ++ ?t2 ++ [TPopAdd]))
```

```
    + apply StoT_RNat.
```

```
(exist (([TPush 3] ++ [TPush 6] ++ [TPopAdd]) ++ ?t2 ++ [TPopAdd]))
```

```
  - apply StoT_ROpp.
```

```
(exist
   (([TPush 3] ++ [TPush 6] ++ [TPopAdd]) ++
    ([TPush 0] ++ ?t ++ [TPopSub]) ++ [TPopAdd]))
```

```
    + apply StoT_RNat.
```

```
(exist
   (([TPush 3] ++ [TPush 6] ++ [TPopAdd]) ++
    ([TPush 0] ++ [TPush 2] ++ [TPopSub]) ++ [TPopAdd]))
```

This shows how the program gets built: each lemma application is equivalent to one recursive call in a run of the compilation function StoT.

Coq has facilities to automatically perform proof search using a set of lemmas, which we can use to automate the derivation of t7: it suffices to register all constructors of ℜ in a "hint database", as follows:[6]

```
Create HintDb cc.
Hint Constructors StoT_rel : cc.

Example t7_rel_eauto: { t7 | t7 ℜ s7 }.
Proof. eauto with cc. Defined.

Compute t7_rel_eauto.
```

```
= exist [TPush 3; TPush 6; TPopAdd; TPush 0; TPush 2; TPopSub; TPopAdd]
: {t7 : T | t7 ℜ s7}
```

And of course, the result is *correct-by-construction*, in the sense that it carries its own proof of correctness:

```
Eval cbn in type of (proj2_sig t7_rel_eauto).
```

```
= [TPush 3; TPush 6; TPopAdd; TPush 0; TPush 2; TPopSub; TPopAdd] ℜ s7
: Type
```

[6]In the following we use eauto as an abbreviation for typeclasses eauto, a more flexible version of Coq's eauto tactic. In particular, using typeclasses eauto allows us to specify priorities on lemmas.

This is traditional *logic programming*, applied to compilers. We have now learned one fact:

> Correctly compiling a program s is the same as proving $\exists\, t, t \sim s$.

3.1.6 Open-ended compilation

The proofs of correctness for the functional version of the compiler (`StoT`, 4) and for the relational version (`StoT_rel`, 5) have the exact same structure. They are both composed of three orthogonal lemmas:

1. `[TPush z] ~ SInt z`
2. `[TPush 0] ++ t ++ [TPopSub] ~ SOpp s`
3. `t1 ++ t2 ++ [TPopAdd] ~ SAdd s1 s2`

Each of these is really a standalone fact, and they each correspond to a *partial* relation between S and T, each connecting *some* programs in S to *some* programs in T. In other words:

> A relational compiler is really just a collection of facts connecting programs in the target language to programs in the source language.

This means that we don't even need to define a relation. Instead, we can have three lemmas that directly refer to the original equivalence `~`:

```
Lemma StoT_Int z :
  [TPush z] ~ SInt z.

Lemma StoT_Opp t s :
  t ~ s →
  [TPush 0] ++ t ++ [TPopSub] ~ SOpp s.

Lemma StoT_Plus t1 s1 t2 s2 :
  t1 ~ s1 →
  t2 ~ s2 →
  t1 ++ t2 ++ [TPopAdd] ~ SAdd s1 s2.
```

And from these, we can build a compiler! We just need to place all these facts into a new database of lemmas, which Coq will use as part of its proof search:

```
Create HintDb c.
Opaque σS σT.
Hint Resolve StoT_Int : c.
Hint Resolve StoT_Opp : c.
Hint Resolve StoT_Plus : c.
```

And then we can derive compiled programs and their proofs:

```
Example t7_fn_eauto: { t7 | t7 ~ s7 }.
Proof. eauto with c. Defined.
Compute t7_fn_eauto.
```

```
= exist [TPush 3; TPush 6; TPopAdd; TPush 0; TPush 2; TPopSub; TPopAdd]
: {t7 : T | t7 ~ s7}
```

Opacity

Making σS and σT *opaque* prevents commands like eauto from unfolding too much of our definitions and being generally to clever. Without it, since our example language S is so simple that all programs are actually just constants (!), our new compiler would just reduce programs before compiling them:

```
Transparent σS σT.
Example t7_fn_eauto_t: { t7 | t7 ~ s7 }. Proof. eauto with c. Defined.
Compute t7_fn_eauto_t.
```

```
= exist [TPush 7]
: {t7 : T | t7 ~ s7}
```

The alternative would be to give priority to StoT_Opp and StoT_Plus over StoT_int, which we demonstrate later in this document.

That is the core idea of relational compilation. On this simple example it looks mostly like an odd curiosity, but it is actually very useful for compiling (shallowly) embedded domain-specific languages (EDSLs), especially when the compiler needs to be extensible.

3.2 Use case 1: Compiling shallowly embedded DSLs

The original set up of the problem (compiling from language S to language T) required us to exhibit a function $f : S \to T$. Not so with the new set up, which instead requires us to prove instances of the ~ relation (one per program). What this means is that we can apply this compilation technique to compile *shallowly* embedded programs, including shallowly embedded DSLs[7].

Here is how we would change our previous example to compile arithmetic expressions written directly in Gallina (Gallina is functional programming language inside of the Coq proof assistant):

1. Start by redefining the relation to use *Gallina expressions* on the right side of the equivalence (there are no more references to S nor σS):

   ```
   Notation "t ≈ s" := (forall ts, σT t ts = s :: ts).
   ```

2. Add compilation lemmas (the proofs are exactly the same as before, so they are omitted). Note that on the right side we have plain Gallina + and -, not SAdd and SOp, so each lemma now relates a shallow program to an equivalent deep-embedded one:

   ```
   Lemma GallinatoT_Z z :
     [TPush z] ≈ z.
   Lemma GallinatoT_Zopp t z :
     t ≈ z ->
     [TPush 0] ++ t ++ [TPopSub] ≈ - z.
   Lemma GallinatoT_Zadd t1 z1 t2 z2 :
     t1 ≈ z1 ->
     t2 ≈ z2 ->
     t1 ++ t2 ++ [TPopAdd] ≈ z1 + z2.
   ```

These lemmas are sufficient to create a small compiler: as before we populate a hint database with our compilation lemmas:

```
Create HintDb stack.
Hint Resolve GallinatoT_Z | 10 : stack.
Hint Resolve GallinatoT_Zopp : stack.
Hint Resolve GallinatoT_Zadd : stack.
```

[7] A shallow embedding is one where programs are defined directly in the host language in contrast with a deep embedding where programs are represented as abstract syntax trees, i.e. data.

And then we run our relational compiler on shallowly embedded input programs:

```
Example g7 := 3 + 6 + Z.opp 2.

Example t7_shallow: { t7 | t7 ≈ g7 }.
Proof. eauto with stack. Defined.

Compute t7_shallow.
```

```
= exist [TPush 7]
: {t7 : T | t7 ≈ g7}
```

Of course, it is easy to package this in a convenient notation (the pattern `match Set return T with _ ⇒ X end` is a roundabout way to force the type of the value `X`):

```
Notation compile gallina_term :=
  (match Set return { t | t ≈ gallina_term } with
   | _ ⇒ ltac:(eauto with stack)
   end)
  (only parsing).

Compute compile (3 + 6 + Z.opp 2).
```

```
= exist [TPush 7]
: {t : T | t ≈ 3 + 6 + - (2)}
```

There is something slightly magical happening here. By rephrasing compilation as a proof search problem, we are been able to make a compiler that would not even be expressible (let alone provable!) as a regular Gallina function. Reasoning on shallowly embedded programs is often much nicer than reasoning on deeply embedded programs, and this technique offers a convenient way to bridge the gap.

3.3 Use case 2: Extensible compilation

Up to this point we assumed that the input language was fixed, but in fact now that we are compiling shallowly embedded Gallina programs we can trivially extend the source language with additional constructs. Fortunately, the relational compilation technique above readily supports extending the compiler to support new source expressions.

In fact, extensible languages is one place where relational compilation shines. As an example, suppose we are modeling combinational hardware circuits in Coq. Our target type (deep-embedded Boolean expressions) is very simple:

```
Inductive circuit :=
| Const (z: Z)
| Read (reg_name: string)
| Mux (cond l r: circuit)
| Op (op_name: string) (args: list circuit).
```

Notice how the `Op` and `Reg` constructors (used to call built-in operators and to read registers) take names as strings: this means that to define an interpreter for the language we need an environment `Σ` of functions defining the semantics of the builtin operators of the language (6) and a context `R` giving the value of each register (7). The code below uses the notation `c.[k]` to look up key `k` in context `c` (it defaults to an arbitrary value if the key `k` cannot be found):

```
Section Interp.
  Variable Σ: string → (list Z → Z). 6
  Variable R: list (string * Z). 7

  Fixpoint cinterp (c: circuit) : Z := match c with
    | Const z ⇒ z
    | Read r ⇒ R.[r]
    | Mux cond t f ⇒
      if cinterp cond =? 0 then cinterp f else cinterp t
    | Op op args ⇒
      Σ op (List.map cinterp args)
  end.
End Interp.
```

Here is an example.

- First, we define an environment of built-in functions:

  ```
  Definition testbit z n :=
    if Z.testbit z n then 1 else 0.

  Example Σ0 fn args := match fn, args with
    | "add", [z1; z2] ⇒ Z.add z1 z2
    | "xor", [z1; z2] ⇒ Z.lxor z1 z2
    | "nth", [z; n] ⇒ testbit z n
    | _, _ ⇒ 0 (* Default to 0 for simplicity *)
  end.
  ```

- Then, we define an environment of registers:

```
Example R0 :=
  [("pc", 16); ("ppc", 14); ("r1", 5); ("r2", 7)].
```

- And finally we can run the interpreter on an example circuit c1:

```
Example c1 :=
  Mux (Op "xor" [Read "pc"; Read "ppc"])
      (Read "r1") (Read "r2").
```

Reducing everything but Σ0 shows the interpretation of Mux in this term:

```
Eval cbn -[Σ0] in cinterp Σ0 R0 c1.
```

```
= if Σ0 "xor" [16; 14] =? 0 then 7 else 5
: Z
```

... and reducing everything gives us the value of this example circuit:

```
Compute cinterp Σ0 R0 c1.
```

```
= 5
: Z
```

Relational compilation is a simple and convenient way to generate such circuits from Gallina expressions. First, we need a compilation relation:

```
Notation "c ≋ g @ Σ // R" := (cinterp Σ R c = g).
```

Then, we need compilation lemmas relating source programs in Gallina and their circuit equivalents.

First, constants:

```
Lemma compile_Const {Σ R} z:
  Const z ≋ z @ Σ // R.
```

Then variable accesses, compiled to register reads:

```
Lemma compile_Read {Σ R} r z:
  z = R.[r] →
  Read r ≋ z @ Σ // R.
```

Then conditionals:

```
Lemma compile_Mux {Σ R} ccond gcond ct gt cf gf:
  ccond ≈ gcond @ Σ // R →
  ct    ≈ gt    @ Σ // R →
  cf    ≈ gf    @ Σ // R →
  Mux ccond ct cf ≈ if gcond =? 0 then gf else gt @ Σ // R.
```

And finally operators. Note that compilation lemmas are parametric on the environment of functions, only requiring that the one function it uses be found in the environment:

```
Lemma compile_add {Σ R} c1 g1 c2 g2:
  (forall x y, Σ "add" [x; y] = Z.add x y) →
  c1 ≈ g1 @ Σ // R →
  c2 ≈ g2 @ Σ // R →
  Op "add" [c1; c2] ≈ Z.add g1 g2 @ Σ // R.

Lemma compile_xor {Σ R} c1 g1 c2 g2:
  (forall x y, Σ "xor" [x; y] = Z.lxor x y) →
  c1 ≈ g1 @ Σ // R →
  c2 ≈ g2 @ Σ // R →
  Op "xor" [c1; c2] ≈ Z.lxor g1 g2 @ Σ // R.

Lemma compile_nth {Σ R} cz gz cn gn:
  (forall z n, Σ "nth" [z; n] = testbit z n) →
  cz ≈ gz @ Σ // R →
  cn ≈ gn @ Σ // R →
  Op "nth" [cz; cn] ≈ testbit gz gn @ Σ // R.
```

That is enough to compile a simple EDSL for circuits. There are a few things worthy of note here; first, we now have a mechanism to refer to bound variables, through the R environment; second, each compilation lemma is parametric on the environment of functions, so the whole compiler is extensible. Let us see these two points in action with a more complex program:

```
Definition gc1 pc ppc (r1 r2: Z) :=
  let correct := pc =? ppc in
  let v := if correct then r1 else r2 in
  testbit v 4.
```

This program checks if its first two arguments to see if they are equal, and returns a different value in each case. The relational compilation goal will looks a bit different from previous ones, because we now need to account for an environment of variables:

```
Example cc1 : { cc1 | forall pc ppc r1 r2,
   cc1 ≋ gc1 pc ppc r1 r2 @ Σ0
   // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)] }.
```

In other words: there exists a circuit cc1 equivalent to the Gallina program gc1 for all inputs pc, ppc, r1, and r2, assuming that these inputs are available in registers and that the circuit runs with the environment of built-ins Σ0 The proof too looks a bit different, because there now are side conditions for certain lemmas:

```
Proof.
  eexists; intros; unfold gc1.
```

```
────────────────────────────────────────────
?cc1 ≋ testbit (if pc =? ppc then r1 else r2) 4 @ Σ0 //
[("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

The program starts with a call to testbit, so we plug in the circuit primitive "nth":

```
  eapply compile_nth.
```

We now get three subgoals: one asserting that we can call the function "nth" given the current function environment Σ, and two corresponding to each of the arguments to testbit in Gallina:

```
  1: reflexivity.
```

Of the two argument subgoals, the first one is a conditional to which none of our compilation lemmas apply: as we defined it, compile_Mux requires a specific Gallina pattern, _ =? 0, which is not present here:

```
────────────────────────────────────────────
?cz ≋ if pc =? ppc then r1 else r2 @ Σ0 //
[("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

```
    apply compile_Mux.
```

```
In environment
pc, ppc, r1, r2 : Z
Unable to unify
 "Mux ?M1440 ?M1442 ?M1444 ≋ if ?M1441 =? 0 then ?M1445 else ?M1443 @ ?Σ // ?R"
with
 "?cz ≋ if pc =? ppc then r1 else r2 @ Σ0 //
  [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]".
```

This is the first instance where the extensibility of relational compilation comes into play. For this example, we can extend the compiler by plugging in a rewrite rule that transforms the program to match a shape supported by the compiler, after which `Compile_Mux` applies:

```
Z_lxor_eqb
     : forall z1 z2, (z1 =? z2) = (Z.lxor z1 z2 =? 0)
```

```
   rewrite Z_lxor_eqb.
```

```
________________________________________________________________
?cz ≈ if Z.lxor pc ppc =? 0 then r1 else r2 @ Σ0 //
[("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

```
   apply compile_Mux.
```

Compiling the conditional leaves us with three more subgoals: one for the test, one for the right branch, and one for the left branch:

```
________________________________________________________________
?ccond ≈ Z.lxor pc ppc @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
________________________________________________________________
?ct ≈ r2 @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
________________________________________________________________
?cf ≈ r1 @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

The first part can be handled using the `compile_xor` lemma:

```
     apply compile_xor; try reflexivity.
```

And this is where the second interesting part of this proof comes about: handling variables. First, let us try something that looks right, but will not work:

```
       apply compile_Const.
```

```
In environment
pc, ppc, r1, r2 : Z
Unable to unify "?c1" with "Const pc" (cannot instantiate
"?c1" because "pc" is not in its scope).
```

It is very important that this shouldn't work, but it is not immediately obvious why it wouldn't. The value pc is indeed *not* a constant, but it you look just at the types, things do line up:

```
────────────────────────────────────────────────────────────
?c1 ≋ pc @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

```
compile_Const pc
     : Const pc ≋ pc @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

The reason it doesn't work is captured in the error message above. When Coq creates the evars denoted by ?..., it associates with each of them a *context* that records which variables they may refer to. Here is, for example, the internal goal that Coq generated for ?c1:

```
─────────────────────────────────────────────────────── c1
circuit
```

In other words, the evar ?c1 cannot refer to *any* of the local variables — which is good, since we're trying to build a closed term! What we want, of course, is compile_Read, which reads into the environment of registers, not compile_Const:

```
      apply compile_Read with (r := "pc"); reflexivity.
```

The same lemma applies to the next three goals, in fact:

```
────────────────────────────────────────────────────────────
?c2 ≋ ppc @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

```
      apply compile_Read with (r := "ppc"); reflexivity.
```

```
────────────────────────────────────────────────────────────
?ct ≋ r2 @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

```
    apply compile_Read with (r := "r2"); reflexivity.
```

```
────────────────────────────────────────────────────────────
?cf ≋ r1 @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

```
    apply compile_Read with (r := "r1"); reflexivity.
```

And finally we have the second argument to the original testbit, the index of the bit that we want to extract:

```
────────────────────────────────────────────────────────────
?cn ≋ 4 @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

```
  apply compile_Const.
Defined.
```

The resulting generated program is as expected, and correct by construction:

```
Print cc1.
```

```
cc1 =
exist
  (Op "nth"
     [Mux (Op "xor" [Read "pc"; Read "ppc"]) (Read "r2") (Read "r1"); Const 4])
     : {cc1 : circuit
       | forall pc ppc r1 r2 : Z,
         cc1 ≋ gc1 pc ppc r1 r2 @ Σ0 //
         [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]}
```

3.3.1 Automating the derivation

The process above is closer to tool-assisted program derivation than to "compilation". Automating it is not hard, but it requires tricks beyond what we have seen above. We will start by creating a hint database to hold our custom compilation lemmas:

```
Create HintDb circuits.
```

For readability, we will use Coq's `Derive` feature to state our compilation goal. `Derive` defines a dependent pair (a term and a proof of a property about it) as two separate names:

```
Derive cc1' SuchThat (forall pc ppc r1 r2,
    cc1' ≋ gc1 pc ppc r1 r2 @ Σ0
    // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)])
  As cc1'ok.
Proof.
  unfold cc1', gc1; clear cc1'.
```

```
________________________________________
forall pc ppc r1 r2 : Z,
?Goal ≋ testbit (if pc =? ppc then r1 else r2) 4 @ Σ0 //
[("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

The first trick we will introduce is forward reasoning. Until now, we have used eauto to pull from a database of lemmas to try to derive a complete proof of a goal. In this mode, eauto is all-or-nothing: it will not make any progress on this goal until we introduce all relevant lemmas:

```
eauto using compile_nth.
```

```
────────────────────────────────────────────
forall pc ppc r1 r2 : Z,
?Goal ≋ testbit (if pc =? ppc then r1 else r2) 4 @ Σ0 //
[("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

This isn't sustainable: we need to guess exactly all lemmas that are required, or nothing happens. Instead we will use a *partial progress* style popularized by [69], in which we program eauto to take a single step and then return to its caller:

```
Hint Extern 2 ⇒ simple apply compile_Const; shelve : circuits.
Hint Extern 2 ⇒ simple apply compile_add; shelve : circuits.
Hint Extern 2 ⇒ simple apply compile_xor; shelve : circuits.
Hint Extern 2 ⇒ simple apply compile_nth; shelve : circuits.
```

Each of these hints allow eauto to *shelve* the current goal after applying the corresponding lemma, which removes the subgoals generated by that lemma from the pool of things the eauto is required to solve and places the on Coq's *shelf*.
These hints are enough to get us started with the derivation of our program. `forward with` ... is a tactic similar to eauto, but it supports partial progress:

```
forward with circuits.
```

```
────────────────────────────────────────────
forall z n : Z, Σ0 "nth" [z; n] = testbit z n
```

```
────────────────────────────────────────────
?cz ≋ if pc =? ppc then r1 else r2 @ Σ0 //
[("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

The first goal we get asks us to prove that we have the right function under the name "nth" in our function context, as before:

```
────────────────────────────────────────────
forall z n : Z, Σ0 "nth" [z; n] = testbit z n
```

```
Hint Extern 1 ⇒ reflexivity : circuits.
forward with circuits.
```

The second goal is where this new approach of partial compilation begins to be useful: we have partially compiled our program, and we can now add more hints to continue making progress. As before, compile_Mux doesn't apply:

The shelf

Coq's *shelf* is a side collection of goals that typically hold terms whose definition will follow from solving other goals. For example, in the following proof, the definition of ?x is on the shelf: it is expected that solving .io#shelf-example.s(eexists).g.ccl will instantiate ?x (note how the call to Show Existentials mentions that ?x is *shelved*):

```
Goal exists x, x + 1 = 2.
  eexists.
```

```
────────────────────────────────────────
?x + 1 = 2
```

```
  Show Existentials.
```

```
Existential 1 = ?x : [ |- Z] (shelved)
Existential 2 = ?Goal : [ |- ?x + 1 = 2]
```

Using the shelf allows us to define forward-reasoning tactics: we use unshelve to bring back the intermediate goals that were shelved within eauto (typeclasses eauto, in fact, since eauto puts things on the wrong shelf — Coq has more than one), and shelve_unifiable to move evars that are not propositional goals back on the shelf:

```
Tactic Notation "step" "with" ident(db) :=
  intros; unshelve typeclasses eauto with db; shelve_unifiable.

Tactic Notation "forward" "with" ident(db) :=
  progress repeat step with db.
```

```
─────────────────────────────────────────────────
?cz ≋ if pc =? ppc then r1 else r2 @ Σ0 //
[("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

```
  apply compile_Mux.
```

```
In environment
pc, ppc, r1, r2 : Z
Unable to unify
 "Mux ?M1502 ?M1504 ?M1506 ≋ if ?M1503 =? 0 then ?M1507 else ?M1505 @ ?Σ // ?R"
with
 "?cz ≋ if pc =? ppc then r1 else r2 @ Σ0 //
  [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]".
```

We could use `Z_lxor_eqb` as before to rewrite the equality test `pc =? ppc` into an exclusive-or test `Z.lxor pc ppc =? 0`, but for consistency we will prove a new compilation lemma instead (this gives us a unified way to handle all extensions):

```
Lemma compile_Mux_eqb {Σ R} c1 g1 c2 g2 ct gt cf gf:
  (forall x y, Σ "xor" [x; y] = Z.lxor x y) →
  c1 ≋ g1 @ Σ // R →
  c2 ≋ g2 @ Σ // R →
  ct ≋ gt @ Σ // R →
  cf ≋ gf @ Σ // R →
  Mux (Op "xor" [c1; c2]) ct cf ≋
  if g1 =? g2 then gf else gt @ Σ // R.
```

This is enough to step further in the compilation process, leaving us with four very similar goals:

```
Hint Extern 2 ⇒
  simple apply compile_Mux_eqb; shelve : circuits.
forward with circuits.
```

This time, we get stuck because we did not register `compile_Read`:

```
─────────────────────────────────────────────────
?c1 ≋ pc @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
─────────────────────────────────────────────────
?c2 ≋ ppc @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
─────────────────────────────────────────────────
?ct ≋ r2 @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

```
______________________________________________________________
?cf ≈ r1 @ Σ0 // [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)]
```

Why not? Because as written, `compile_Read` applies to all goals, often leaving an unsolvable goal: when applied to a goal `_ ≈ g @ _ // R`, `compile_Read` will simply generate a goal asking to find `g` in `R`, regardless of whether such a binding in fact exists in `R`. For example:

```
eexists ?[c].
```

```
______________________________________________________________
?c ≈ 1 + 1 @ Σ0 // [("r0", 14)]
```

```
apply compile_Read.
```

```
______________________________________________________________
1 + 1 = [("r0", 14)].[?r]
```

```
Abort.
```

There are two ways to proceed in such cases: add logic to apply `compile_Read` eagerly, and backtrack if no corresponding binding can be found; or use a more clever strategy to apply `compile_Read` more discerningly. Up to this point our derivations have all been deterministic, and we want the compiler to be as predictable as possible, so we will do the latter by restricting the use of the `compile_Read` lemma to cases where the Gallina term `g` is a single variable. Additionally, we will give `compile_Read` a low priority (the backtracking approach is discussed in a note at the end of this section):

```
Hint Extern 3 (_ ≈ ?v @ _ // _) ⇒
  is_var v; simple apply compile_Read; shelve : circuits.

all: forward with circuits.
```

The remaining goals are the preconditions of `compile_Read`: the variables should in fact be in context:

```
______________________________________________________________
pc = [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)].[?r]

______________________________________________________________
ppc = [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)].[?r0]

______________________________________________________________
r2 = [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)].[?r1]

______________________________________________________________
r1 = [("pc", pc); ("ppc", ppc); ("r1", r1); ("r2", r2)].[?r2]
```

To solve these goals automatically, we will use two simple lemmas.

1. assoc_hd, which handles the case in which the value we're looking for is the first in the list:

   ```
   Lemma assoc_hd (v: V) k tl:
     v = ((k, v) :: tl).[k].
   ```

2. assoc_tl, which handles the case in which the value we're looking for is in the tail of the list:

   ```
   Lemma assoc_tl (v v': V) k k' tl:
     v = tl.[k'] →
     k ≠ k' →
     v = ((k, v') :: tl).[k'].
   ```

Here is how these come into play, on a standalone example. We start with a goal asking us to locate the value 3 in a context. Applying assoc_tl discards the first binding ("x", 1) and asks us to find 3 among the remaining bindings; then, applying assoc_hd selects the first of the remaining bindings ("y", 3):

```
────────────────────────────────────────
3 = [("x", 1); ("y", 3)].[?k]
```

```
apply assoc_tl.
```

```
────────────────────────────────────────
3 = [("y", 3)].[?k]
────────────────────────────────────────
"x" ≠ ?k
```

```
- apply assoc_hd.
- congruence.
```

Plugging in these two lemmas completes the derivation:

```
  Hint Extern 1 ⇒ congruence : circuits.
  Hint Resolve assoc_hd assoc_tl | 2 : circuits.
  all: forward with circuits.
Qed.

Print cc1'.
```

```
cc1' =
Op "nth"
  [Mux (Op "xor" [Read "pc"; Read "ppc"]) (Read "r2") (Read "r1"); Const 4]
     : circuit
```

3.3.1.1 Automating the initial setup

This handles the derivation itself; the initial unfolding phase of the derivation can be also handled automatically using a simple *Ltac* script to reveal the evar created by `Derive` and unfold toplevel program definitions, like `gc1` above; we do all this in a new tactic `compile with <database>`:

```
Tactic Notation "setup" "with" ident(db) :=
  intros; autounfold with db;
  lazymatch goal with c := _ ⊢ _ ⇒ subst c end.
Tactic Notation "compile" "with" ident(db) :=
  setup with db; forward with db.
```

And with this, we have our first, tiny, extensible compiler! Of course, this new compiler is applicable to a wide range of programs, not just the one we just compiled:

```
Derive c SuchThat (forall z, c ≋ z + z @ Σ0 // [("z", z)]) As cok.
Proof. compile with circuits. Qed.
Print c.
```

```
c = Op "add" [Read "z"; Read "z"]
     : circuit
```

3.3.2 Extending the compiler

There are all sorts of ways we can extend this compiler; the following are just a few examples:

3.3.2.1 Compiling open terms (macros)

Because the compilation process does not have to start into an empty environment, we can compile macros, not just functions: all that is needed is to compile the function with an indeterminate function environment and indeterminate registers. We will allow ourselves to call the `add` function (through `Hadd`), as well as an arbitrary precompiled program `c` (through `Hc`):

```
Context Σ R c z
        (Hadd: forall x y, Σ "add" [x; y] = x + y)
        (Hc: c ≋ z @ Σ // R).
Derive c3 SuchThat (c3 ≋ z + z + z @ Σ // R) As c3ok.
```

```
Proof. compile with circuits. Qed.

Print c3.
```

```
c3 = Op "add" [Op "add" [c; c]; c]
     : circuit
```

After that, the c3 macro can be called automatically where appropriate by adding a compilation hint:

```
Hint Extern 2 ⇒ simple apply c3ok : circuits.

Derive c6 SuchThat (c6 ≋ (z+z+z) + (z+z+z) @ Σ // R) As c6ok.
Proof. compile with circuits. Qed.

Print c6.
```

```
c6 = Op "add" [c3; c3]
     : circuit
```

3.3.2.2 Plugging in new builtins

We can make use of additional functions by extending the function environment Σ, which defines the semantics of builtins, and adding appropriate compilation lemmas:

```
Lemma compile_lsl {Σ R} c1 g1 c2 g2:
  (forall x y, Σ "lsl" [x; y] = x ≪ y) →
  c1 ≋ g1 @ Σ // R → c2 ≋ g2 @ Σ // R →
  Op "lsl" [c1; c2] ≋ g1 ≪ g2 @ Σ // R.
Proof. cbn; repeat intros →; reflexivity. Qed.

Hint Extern 2 ⇒ simple apply compile_lsl; shelve : circuits.

Example Σ1 fn args := match fn, args with
  | "lsl", [z1; z2] ⇒ z1 ≪ z2
  | _, _ ⇒ Σ0 fn args
end.

Derive c4 SuchThat (forall z, c4 ≋ z ≪ 2 @ Σ1 // [("z", z)]) As c4ok.
Proof. compile with circuits. Qed.

Print c4.
```

```
c4 = Op "lsl" [Read "z"; Const 2]
     : circuit
```

3.3.2.3 Using custom logic to prove side-conditions

Instead of the two lemmas that we proved earlier for compile_Read side-conditions (assoc_hd and assoc_tl), we can use a custom metaprogram (a *tactic*) to figure out the right variable name and plug it right in. For that, we need a tactic that perform a reverse lookup in an association list, defined below by induction:

```
Ltac assocv v ctx :=
  lazymatch ctx with
  | []             ⇒ fail
  | (?k, v) :: _    ⇒ k
  |        _ :: ?ctx ⇒ assocv v ctx
  | _ ⇒ let ctx := eval red in ctx in assocv v ctx
  end.

Compute assocv 14 [("x", 3); ("y", 14)].
```

```
= "y"
: string
```

And using assocv we can define a tactic that guesses the right variable name ?k in goals like v = ctx.[?k]. In the example below, instantiate_assoc_eq guesses "y" for the value "3":

```
Ltac instantiate_assoc_eq := match goal with
  | ⊢ ?v = ?ctx.[?k] ⇒
    is_evar k; let k0 := assocv v ctx in unify k k0
end.
```

```
────────────────────────────────────────
3 = [("x", 1); ("y", 3)].[?k]
```

```
  instantiate_assoc_eq.
```

```
────────────────────────────────────────
3 = [("x", 1); ("y", 3)].["y"]
```

And finally we add a hook in the compiler to use that tactic as appropriate:

```
Hint Extern 1 ⇒ instantiate_assoc_eq : circuits.

Derive cc2 SuchThat
  (forall x y, cc2 ≋ x + y @ Σ0 // [("x", x); ("y", y)])
As cc2ok. compile with circuits. Qed.
```

```
Print cc2.
```

```
cc2 = Op "add" [Read "x"; Read "y"]
     : circuit
```

3.3.3 Leveraging contextual information

The last extension is important enough that it deserves its own section. It is a common pattern in functional programming languages to use a monad to describe an effect that is not supported by the language. Part of compiling the program down to a lower-level language is mapping the monadic structure to low-level primitives, and we can do that using relational compilation.

Even better, we can support native compilation of arbitrary monads (!): unlike traditional compilers that hard-code a list of built-in monads that get special compiler support (think `IO` in Haskell), we can plug-in native compilation support for arbitrary user-defined monads. This means that we never have to define an interpreter for a free monad (this is the usual approach in Coq for extraction effectful program: define a free monad specialized to a functor capturing effects, extract to OCaml, and define an unverified interpreter to map the effects to native OCaml effects; here, we can instead directly map the monad to effects of the target language).

We will start by exploring the example of the `Writer` monad with a compiler specialized for that monad, and then we will see an extra twist that allows us to define the pure parts of the compiler in a monad-agnostic way, so that they can be shared between different EDSL compilers specialized to different monads, reducing duplication. Rupicola itself has many more examples of relational compilation for monadic programs.

3.3.3.1 The writer monad

In this example, programs will return not just values, but also a list of strings, the "output" of the program:

```
Definition Trace := list string.
Record S {α: Type} := { val: α; trace: Trace }.

Example puts str := {| val := tt; trace := [str] |}.
```

The usual monadic operators are readily defined: a pure computation is like a monadic computation with an empty trace, and two effectful computations running in sequence produce the result of the second and the concatenation of the two traces:

```
Definition ret (a: α) : S α :=
  {| val := a; trace := [] |}.

Definition tr_bind (a: S α) (b: S β) :=
  {| val := b.(val); trace := a.(trace) ++ b.(trace) |}.

Definition bind (a: S α) (k: α → S β) : S β :=
  tr_bind a (k a.(val)).

Notation "v ← a ; body" := (bind a%string (fun v ⇒ body)).
```

It is straightforward to prove that the usual monad properties hold:

```
Lemma bind_ret (ca: S α) :
  bind ca ret = ca.
Lemma ret_bind a (k: α → S β) :
  bind (ret a) k = (k a).
Lemma bind_bind ca (ka: α → S β) (kb: β → S γ) :
  bind (bind ca ka) kb = bind ca (fun a ⇒ bind (ka a) kb).
```

We will compile this to a simple imperative string language with traces. Here is the definition of expressions and statements in that language; this will also give us the opportunity to start discussing assignments:

```
Inductive Expr :=              Inductive T :=
| Ref var                      | Seq (t1 t2: T)
| Const str                    | Assign var (e: Expr)
                               | Puts (e: Expr)
| Concat (e1 e2: Expr).
                               | Skip.
```

The semantics of expressions is given by an interpreter:

```
Definition Ctx := list (Var * string).

Fixpoint interp ctx e : string := match e with
  | Ref var      ⇒ ctx.[var]
  | Const str    ⇒ str
  | Concat e1 e2 ⇒ interp ctx e1 ++ interp ctx e2
end.
```

… and, to spice things up, the semantics of statements is given by a big-step evaluation relation:

```
Inductive RunsTo : Ctx → T → Ctx → Trace → Prop :=
| RunsToSeq ctx0 t1 ctx1 tr1 t2 ctx2 tr2:
    ⟨ctx0, t1⟩ ⇓ ⟨ctx1, tr1⟩ →
    ⟨ctx1, t2⟩ ⇓ ⟨ctx2, tr2⟩ →
    ⟨ctx0, Seq t1 t2⟩ ⇓ ⟨ctx2, tr1 ++ tr2⟩
| RunsToAssign ctx var e:
    ⟨ctx, Assign var e⟩ ⇓ ⟨(var, interp ctx e) :: ctx, []⟩
| RunsToPuts ctx e:
    ⟨ctx, Puts e⟩ ⇓ ⟨ctx, [interp ctx e]⟩
| RunsToSkip ctx e:
    ⟨ctx, Skip⟩ ⇓ ⟨ctx, []⟩
where "⟨ ctx , p ⟩ ⇓ ⟨ ctx' , tr ⟩" :=
  (RunsTo ctx p ctx' tr).
```

The relational compiler for expressions is routine at this point, so we omit it for brevity. We simply define a relation $\sim_e$ for expressions, and prove lemmas relating outputs and inputs of interp. For very simple cases like this one, what we are really building is in fact a reification procedure, implemented by programming a decision procedure to invert the function interp:

```
Notation "e ~e g // ctx" := (interp ctx e = g%string).

Derive eHello SuchThat (forall name,
  eHello ~e ("hello, " ++ name) // [("name", name)])
As eHello_ok. compile with str. Qed.

Print eHello.
```

```
eHello = Concat (Const "hello, ") (Ref "name")
     : Expr
```

Conversely, there are a few new things in the relational compiler for statements. Specifically, we want to convert uses of monadic bind into a sequence, translating pure expressions using the expression compiler and using primitives to implement stateful computations like puts.

There is one significant difference from previous examples, however: our new target language has assignments, and these assignments are not directly reflected in the source language. As a result the final state of the program may contain arbitrary bindings, and it would be quite inconvenient to have to declare exactly which temporary variables may be used when starting the compilation process. Instead, we will use a slightly more complicated compilation relation. Unlike the equalities used previously, we now state that a low-level program is related to a high-level one if they produce the same traces, and if the result of the high-level program can be found in the final context, under a name specified as part of the compilation relation. Beyond this, the final context is allowed to contain arbitrary bindings.

```
Definition related t g var ctx :=
  (forall ctx' tr,
     ⟨ ctx, t ⟩ ⇓ ⟨ ctx', tr ⟩ →
     g.(trace) = tr ∧
     g.(val) = ctx'.[var]).

Notation "t ~t g @ var // ctx" :=
  (related t g%string var ctx).
```

Each lemma about the new compilation relation $\sim_t$ matches a corresponding constructor of RunsTo closely, but not exactly, because we need to phrase things in terms of monadic operations. First we show how to compile Puts, which writes a value out:

```
Lemma compile_Puts ctx e g t k var:
  e ~e g // ctx →
  t ~t k tt @ var // ctx →
  Seq (Puts e) t ~t bind (puts g) k @ var // ctx.
```

Then Assign (quantifying over the value v prevents the expression g from getting inlined into the continuation k):

```
Lemma compile_Assign ctx e g t k var tmp:
  e ~e g // ctx →
  (forall v, v = g → t ~t k v @ var // (tmp, v) :: ctx) →
  Seq (Assign tmp e) t ~t bind (ret g) k @ var // ctx.
```

And finally a lemma to conclude the compilation, which uses an assignment because the way our compilation relation is phrased (see sidebar).

```
Lemma compile_Skip ctx e str var:
  e ~e str // ctx →
  Assign var e ~t ret str @ var // ctx.
```

These compilation rules are enough to compile full programs (below, the tactic binder_name translates a Coq-level binder name into a string):

```
Hint Extern 1 ⇒ simple apply compile_Puts; shelve : str.
Hint Extern 1 ⇒ simple apply compile_Skip; shelve : str.

Hint Extern 1 (_ ~t bind ?s ?k @ _ // _) ⇒
  simple apply compile_Assign
    with (tmp := ltac:(binder_name k)); shelve : str.

Definition greet (name: string) :=
  greeting ← ret ("hello, " ++ name);
  _ ← puts greeting;
  _ ← puts "!";
  ret greeting.
Hint Unfold greet : str.

Derive tHello SuchThat (forall name,
  tHello ~t greet name @ "out" // [("name", name)])
As tHello_ok. compile with str. Qed.

Print tHello.
```

```
tHello =
Seq (Assign "greeting" (Concat (Const "hello, ") (Ref "name")))
  (Seq (Puts (Ref "greeting"))
     (Seq (Puts (Const "!")) (Assign "out" (Ref "greeting"))))
     : T
```

3.3.3.2 Monad-agnostic extraction

In the above, we defined a new extraction procedure for each monad, but we can reduce the required effort by generalizing further. We can define things in such a way that lemmas about non-monadic code work for all monad, which means that different domain-specific languages, using different monads, can all use the same code for compiling pure values. The key here is to completely generalize over the pre and post-conditions that the compiler uses, leaving only a distinguished argument to the postcondition indicating which program we are compiling. For this we define a Hoare triple on top of our program semantics (for brevity we define it on top of our big-step semantics instead of defining a new relation, and we omit the precondition, which will live in the ambient proof context instead):

```
Definition triple {α}
           (ctx: Ctx) (prog: T) (spec: α)
           (post: α → Trace → Ctx → Prop) :=
  forall ctx' tr,
    ⟨ctx, prog⟩ ⇓ ⟨ctx', tr⟩ → post spec tr ctx'.
```

```
Notation "<{ ctx }> prog <{ spec | post }>" :=
  (triple ctx prog spec post).
```

Note how the post-condition post takes a special argument spec, which is where we will plug in our Gallina programs (this trick makes it easy to spot the program that we're compiling when writing tactics that inspect the goal). Crucially, spec does *not* take a monadic argument in: any Gallina program is fair game to plug into this spot that drives the compiler. Our previous relation is a special case of this one, and in fact all lemmas will carry naturally. Specifically, we have:

```
Goal forall t g var ctx,
    <{ ctx }>
    t
    <{ g | fun g tr' ctx' ⇒
         g.(val) = ctx'.[var] ∧
         g.(trace) = tr' }> →
    t ~ₜ g @ var // ctx.
```

Here is compile_Assign written in this new style. We quantify the hypothesis about expressions over all states allowed by the precondition, and as the postcondition we plug in the strongest postcondition for the assignment statement (which looks simpler than the usual SP of an assignment because of the choice to move preconditions to the surrounding logic).

```
Lemma compile_Assign e g ctx tmp:
  e ~ₑ g // ctx →
  <{ ctx }>
    Assign tmp e
  <{ g | fun v tr ctx' ⇒
       tr = [] ∧ ctx' = (tmp, v) :: ctx }>.
```

And with these generalized pre-post pairs, we can now define a compilation lemma for bind, as well as a proper Skip lemma, both of which vexed us previously; this time the first program is compiled with an arbitrary postcondition, with will be resolved through unification as part of the compilation process; and the second program assumes this intermediate postcondition as its starting point and completes its run with a modified postcondition that appends the corresponding trace. Because this lemma *does* mention low-level effects, of course, we do need to mention the monad that we use to implement trace-modifying effects.

```
Lemma compile_Seq {α β} ctx post middle p1 p2 (s1: S α) (s2: α → S β):
  <{ ctx }> p1 <{ s1 | middle }> →
  (forall g1 tr1 ctx1, g1 = s1 → middle g1 tr1 ctx1 →
   let post' g2 tr2 := post (tr_bind g1 g2) (tr1 ++ tr2) in
   <{ ctx1 }> p2 <{ s2 g1.(val) | post' }>) →
  <{ ctx }> Seq p1 p2 <{ bind s1 s2 | post }>.

Lemma compile_Skip g ctx post :
  post g [] ctx →
  <{ ctx }> Skip <{ g | post }>.
```

Conditionals and loops can be handled similarly to sequences. For straightline code, we do not need to instantiate this "middle" clause explicitly: instead we can simply let it be derived by unification as part of compiling the first part of the sequence. For conditionals and for loops there is no free lunch, however, so we need to infer a predicate that captures the effect of both branches (for conditionals) or arbitrary repetitions (for loops). Luckily this inference problem is easier than it seems at first: specifically, we can pick the strongest postcondition, with carefully chosen heuristics and manipulations to ensure that we chose a postcondition that is readable and workable for the rest of the compilation process. The exact choice of heuristics is out of scope for this section, but we detail it later when we dive into the specifics of Rupicola (section 4.5.2).

At this point, if we consider the case of a source program made of a sequence of let-bindings, we realize that the compilation process will now be an alternation of `compile_Seq` and specialized compilation lemmas, the former introducing cuts in the derivation and the latter refining the precondition of the program. Once we're done with all let-bindings (or monadic binds), we unify the final precondition that the compiler has derived with the postcondition that we were hoping to achieve.

It is because of this last step that we usually prefer to add an explicit continuation to each lemma, even though our new representation allows for a general cut lemma `compile_Seq`: making continuations explicit allows us to craft our intermediate postconditions very precisely as part of writing each lemma, instead of relying on unification. (Careful readers will have noticed the difficulty already popping up in a small way in `compile_Assign` above, which referred to a variable name `tmp` that we could not infer from goal, since we had already eliminated the corresponding let binding.) This makes the last unification step trivial in almost all cases. Here is what `compile_Assign` looks like in this style (using a wrapper `blet` — for "blocked let" — around let-bindings to prevent Coq from unfolding too aggressively without depending on a specific monad and bind + ret):

```
Lemma compile_Assign {α} ctx e g t (k: string → α) tmp post:
  e ~ₑ g // ctx →
  (forall v, v = g → <{ (tmp, v) :: ctx }> t <{ k v | post }>) →
  <{ ctx }> Seq (Assign tmp e) t <{ blet g k | post }>.
Proof. unfold triple; hammer. Qed.
```

This is the last step of our journey, and since our final compiler looks so similar to the previous iteration, we omit it for brevity; curious readers can consult the complete development for details.

Backtracking

In my experience, it is almost always best to reduce or eliminate backtracking. Crafting compilers is already a tricky business, and introducing backtracking makes debugging significantly harder. Generating proofs along programs ensure that the compiler does not produce *incorrect* output, but it does not rule out *undesirable* output, such as inefficient programs.

Still, in the `compile_Read` case above, an alternative approach would have been to apply the lemma unconditionally, but *without shelving*: in that case `eauto` would only have applied it when it is able to solve the resulting assoc subgoals:

```
Hint Resolve compile_Read assoc_hd assoc_tl : circuits.
```

No rules arbitrary sequences (no cuts)

It may be surprising that we do not have a lemma for compiling an arbitrary sequence — one that would handle any Gallina `bind`. Instead, we have two lemmas, specialized to the two kinds of expressions that may be bound, `Puts` and `Assign`, both with an arbitrary continuation. Is that not limiting? Not really.

The reason we do not have an arbitrary bind is that our equivalence relation is not precise enough: it does not specify exactly what the environment of variables is after running a piece of code. This is convenient, since it allows to phrase our lemmas concisely and easily; but it prevents us from phrasing a lemma to relate `bind` and `Seq` in general (and it makes it so that our `Skip` lemma does not in fact use `Skip`). What would such a bind lemma look like? We might hope to write something like the following, but this is not valid:

```
Lemma compile_Seq ctx t1 s1 t2 s2 var tmp:
  t1 ~t s1 @ tmp // ctx →
  t2 ~t s2 s1.(val) @ var // (tmp, s1.(val)) :: ctx →
  Seq t1 t2 ~t bind s1 s2 @ var // ctx.
Abort.
```

Indeed, the first premise says very little about the state of the machine after running `t1`: only that it will contain a binding `tmp ↦ s1.(val)`. It does not say that `ctx` will be unmodified otherwise (`t1` may create new bindings), and hence the environment in which `Seq t1 t2` will run `t2` is not guaranteed to be `(tmp, s1.(val)) :: ctx`, as required by the second premise.

This is not an issue for straightline code, but it does cause trouble for conditionals and loops that appear in the value part of a `bind` (what local variables can we assume the environment to contain after an `if`?). The monad-agnostic approach that we present in the next step solves this problem.

4 Relational compilation for performance-critical applications

The first part of this thesis presented the key ideas behind relational compilation, keeping implementation details to a minimum. In this part I discuss how these ideas come together to implement a realistic compiler-construction toolkit. Specifically, I present the design and implementation of Rupicola, a Gallina-to-Bedrock2 compiler-construction framework with a focus on simple, low-level performance-critical programs.

Rupicola's core is very small (hundreds of lines), but thanks to a variety of extensions the whole distribution end up with a reasonably expressive input language. With all extensions loaded, Rupicola supports arithmetic over many types (Booleans, bounded and unbounded natural numbers, bytes, integers, machine words), various control-flow patterns (conditionals as well as iteration patterns like maps and folds, with and without early exits), various flat data structures such as mutable cells and arrays; plain and monadic binds; various monadic extensions including the nondeterminism, writer, and I/O monads and a generic free monad; and various low-level effects and features such as stack allocation, inline tables, intrinsics, and external functional calls (details about these features are given in section 4.6.2)

For the programs that fit within Rupicola's existing input language, using our compiler is not very different from using any other (research-quality) compiler: plug in a function body and a signature and get compiled output back. For programs that Rupicola does not support out of the box, or for programs that require custom reasoning to jump the functional-to-imperative gap (typically array-bounds side conditions), users are able to plug in new compilation lemmas or new logic at a reasonable cost.

I designed the framework so that the default reaction to unexpected input would be to stop and ask for user guidance rather than default to a slower generic implementation. As a result, Rupicola makes few guesses and consequently very few *incorrect* guesses: programs compiled using Rupicola achieve performance comparable to that of equivalent handwritten C programs, because Rupicola produces C programs that are (semantically[8]) very close to handwritten C programs.

I start with a description of Rupicola's input and output languages: subsets of Gallina, the functional language of the Coq proof assistant, for inputs; and Bedrock2, a low-level programming and verification language developed at MIT, for outputs. I then review the high-level architecture of Rupicola, combining a minimal compiler core and various compilation domains as orthogonal extensions. Then, I dive deep on two new ideas that make Rupicola possible, showcasing the unique challenges each of them pose and explaining how I tackled them.

Before all that, however, let us get a taste of programming in Rupicola by revisiting our introductory example (uppercasing a string).

4.1 Compiling with Rupicola

Recall the setup: we are programming some sort of web service, and part of the work of resolving a query is to change an input string to uppercase. Gallina represents strings using a datatype similar to a linked list of characters, and ASCII characters as 8-tuples of Booleans (bits). On top of these definitions we implement a recursive `String.map` to apply a transformation to each individual character, and we pass it a new function `toupper` implemented using a `match` mapping each lowercase ASCII character to its uppercase counterpart. There are four main ways in which we want the final, compiled implementation of this program to differ from the execution strategy suggested by the high-level program, and accordingly we will need to combine four compiler extensions to translate this program to Bedrock2, the C-like language that Rupicola targets.

- First, we want to change the way strings and characters are represented: we want strings to be contiguous arrays of characters, not linked lists, and we want characters to be machine bytes, not structs with eight single-bit fields.
- Second, we want to get rid of higher-order iteration: we would like to translate the string map to a loop, thereby reducing pressure on the stack and memory.

[8] Rupicola uses Bedrock2's pretty-printer to C, whose output is not optimized for readability and does not use the whole range of C types.

- Third, we want to introduce mutation: the source program is pure, but the result should modify the string in-place.
- Fourth, we want to optimize the calculation of the per-character uppercasing operation, using specific properties of the ASCII encoding that we use to represent characters.

Some of these transformations are expressible at source level, but not all, or at least not equally easily. For example, Gallina, the language that we start with, defines strings as linked lists of 8-Boolean records, but we can define our own pure array, chars, and string types; Gallina has no loop built-ins, but we can define higher-order iterators that simulate low-level loops; Gallina does not natively support mutation, but we could rewrite the program using the state monad; and the bit and number tricks that we want to perform on ASCII characters to uppercase them are not easy to express on 8-Boolean records, but we can cast to and from a better representation. And yet, even if we performed all that encoding work, we would still have no way using Coq's native extraction to generate code that reliably matches handwritten-C performance.

Rupicola solves the problem nicely. The four implementation choices are domain- or program-specific design choices, and Rupicola enables the user to leverage a mix of source-level annotations, transformations, and compiler extensions to communicate them to the compiler.

In the simplest cases, including this one, source-level annotations are not necessary: judiciously chosen compiler hints suffice to derive low-level code directly from an unmodified high-level representation. In more complex cases, or in cases requiring more precise control of the output, users instead provide a "low-level Gallina" version of the program directly suitable for compilation to Bedrock2, and separately relate that functional model to a high-level specification. This second approach is the most common, so that is the one that I demonstrate here.

Let us start from a high-level specification of the problem:

```
Definition toupper (c: ascii) :=
  match c with
  | "a" ⇒ "A" | "b" ⇒ "B" | "c" ⇒ "C" | "d" ⇒ "D"
  | "e" ⇒ "E" | "f" ⇒ "F" | "g" ⇒ "G" | "h" ⇒ "H"
  | "i" ⇒ "I" | "j" ⇒ "J" | "k" ⇒ "K" | "l" ⇒ "L"
  | "m" ⇒ "M" | "n" ⇒ "N" | "o" ⇒ "O" | "p" ⇒ "P"
  | "q" ⇒ "Q" | "r" ⇒ "R" | "s" ⇒ "S" | "t" ⇒ "T"
  | "u" ⇒ "U" | "v" ⇒ "V" | "w" ⇒ "W" | "x" ⇒ "X"
  | "y" ⇒ "Y" | "z" ⇒ "Z" | c ⇒ c
  end%char.

Definition upstr (s: string) :=
  String.map toupper s.

Compute upstr "rupicola".
```

```
= "RUPICOLA"
: string
```

We then define a lowered version of the same code. Because of the simplicity of this example, the lowering from the natural high-level version is almost trivial: it suffices to define a variant upstr' of upstr on the type list byte instead of string, using the ListArray.map iterator instead of String.map:

```
Definition upstr' (s: list byte) :=
  let/n s := ListArray.map
               (fun b ⇒ byte_of_ascii (toupper (ascii_of_byte b)))
               s in
  s.
```

The equivalence proof is a matter of just a few lines:

```
Lemma string_map_is_map f s:
  String.map f s =
  string_of_list_ascii (List.map f (list_ascii_of_string s)).
Proof. induction s; simpl; congruence. Qed.

Hint Unfold nlet upstr upstr'
            list_byte_of_string string_of_list_byte : lowering.
Hint Rewrite string_map_is_map map_map
             list_ascii_of_string_of_list_ascii : lowering.

Lemma upstr_ok bs:
  list_byte_of_string (upstr (string_of_list_byte bs)) = upstr' bs.
Proof.
  autounfold with lowering; autorewrite with lowering;
    reflexivity.
Qed.
```

And with that, we are ready to start assembling a compiler.

- The first transformation (strings as arrays, chars as bytes) we encode as part of the precondition of our low-level program: we state that we have a pointer s_ptr to a buffer containing the same data as the string. The postcondition is written as translating the string into a list of bytes and invoking the array predicate. The corresponding specification is shown below; it helps to see it as a signature (or a calling convention) for the low-level program that we intend to generate. The function takes two arguments (machine words) p (a pointer to a string) and wlen (its length as a number) and a ghost argument s (a list of bytes), and it returns nothing; the requires clause specifies how the function is called (with a condition on argument wlen and a separation-logic predicate, plus a condition to ensure that all elements of the array are addressable[9]); and the ensures clause states that the program does not produce observable I/O (tr' = tr) and that it overwrites the original string with the update one (upstr' s).

```
Notation nbytes := (sizedlistarray_value AccessByte).
Instance spec_of_upstr : spec_of "upstr" :=
  fnspec! "upstr" p wlen / (s: list byte) r, {
    requires tr mem :=
      wlen = of_nat (length s) ∧
      Z.of_nat (length s) < 2 ^ 64 ∧
      (nbytes (length s) p s ⋆ r) mem;
    ensures tr' mem' :=
      tr' = tr ∧
      (nbytes (length s) p (upstr' s) ⋆ r) mem'
  }.
```

- The second transformation (map as a loop) is done using a lemma to translate ListArray.map into a for loop. This sort of translation is a common pattern, so Rupicola's standard library has built-in support for it: it suffices to import the corresponding compilation module, and to register a hint to solve indexing-related side-conditions.

```
Import LoopCompiler.
Hint Extern 1 => lia : compiler_side_conditions.
```

- The third transformation (mutation) comes as a side effect of using an in-place map-to-loop lemma; in Rupicola I call it an intensional effect, since it is introduced automatically

[9]The nbytes separation logic precondition implicitly guarantees that the s has at most 2^{64} characters (length s ≤ 2^64), indexed from 0 to $2^{64} - 1$. Frustratingly, this is not sufficient iterate over the array: for that we need the length of the array to be representable as a 64-bits integer, i.e. length s < 2^64 - 1, which requires a separate premise.

by analyzing the source code (and not explicitly encoded using a monad). All we need is to give guidance to the compiler on how to compile array operations, which we do by importing a standard library module:

```
Import SizedListArrayCompiler.
```

- The last transformation, efficient uppercasing, can be plugged into the compiler as a rewrite: we prove a program equivalence between our toupper function on 8-tuples of Booleans and an efficient byte computation:

  ```
  Definition toupper' (b: byte) :=
    if byte.wrap (b - "a") <? 26
    then byte.and b x5f else b.
  ```

  Once again the proof is trivial, and we can plug the resulting lemma into the compiler as a rewrite rule, followed by an unfolding rule to allow it to inline upchar':

  ```
  Lemma toupper'_ok b:
    byte_of_ascii (toupper (ascii_of_byte b)) = toupper' b.
  Proof. destruct b; reflexivity. Qed.

  Hint Rewrite toupper'_ok : compiler_cleanup.

  Hint Unfold toupper' : compiler_cleanup.
  ```

With these four pieces in place we can invoke the compiler, and we get the expected low-level program out with no further manual intervention:

```
Derive upstr_br2fn SuchThat
  (defn! "upstr"("s", "len") { upstr_br2fn },
   implements upstr') As upstr_br2fn_ok.
Proof. compile. Qed.
```

The result is a Bedrock2 program upstr_br2fn and its proof of correctness upstr_br2fn_ok; the program can then be compiled using Bedrock2's verified compiler (with support for linking against separately verified fragments of RISC-V machine code as needed, though this example is self-contained), or it can be pretty-printed to C and fed to a traditional C compiler (the details of Bedrock2's pretty-printer are discussed in section 5.2.1):

```
void upstr(uintptr_t s, uintptr_t len) {
  uintptr_t _gs_to, _gs_tmp, _gs_from;
  _gs_from = (uintptr_t)0ULL;
  _gs_to = len;
  while ((_gs_from)<(_gs_to)) {
    _gs_tmp = _br2_load((uintptr_t)(((char*)(s))+(((uintptr_t)1ULL)*(_gs_from))), 1);
    if ((((_gs_tmp)-((uintptr_t)97ULL))&((uintptr_t)255ULL))<((uintptr_t)26ULL)) {
      _gs_tmp = (_gs_tmp)&((uintptr_t)95ULL);
    } else {
      /*skip*/
    }
    _br2_store((uintptr_t)(((char*)(s))+(((uintptr_t)1ULL)*(_gs_from))), _gs_tmp, 1);
    // unset _gs_tmp
    _gs_from = (_gs_from)+((uintptr_t)1ULL);
  }
  return;
}
```

4.2 Rupicola's target language: Bedrock2

Rupicola compiles to Bedrock2 [14], an untyped version of the C programming language. It has a verified compiler to RISC-V with a complete correctness proof as well as a minimal program logic. The semantics divide the program state in three parts: the heap (a flat array of bytes indexed by natural numbers, with an optional layer of separation logic in the program logic), the current function context (a map of names to machine words), and an event trace capturing externally observable events.

Bedrock2's structured control flow includes function calls, conditionals, and loops; the semantics only give meaning to terminating loops, so proofs about Bedrock2 programs are total-correctness proofs. Additionally, stack usage is measured and restricted, so there is no general recursion. Memory allocation is handled by client code, except for allocation on the stack, which is available through a language primitive that gives client code access to temporary scratch space that is lexically scoped within a function's body. Finally, Bedrock2 has no global constants except through the heap, but it has a language primitive for static arrays of byte constants (inline tables).

The choice of Bedrock2 was mostly one of convenience, along with consideration for the long-term goal of end-to-end verification. Bedrock2's flat-array-of-bytes model of memory lends itself nicely to writing efficient programs, too, since it makes it easy to program patterns that are hard to write in C (for example, C makes it particularly difficult to reinterpret pointers, e.g. to iterate over an array of one type as an array of another type, or to reinterpret a structure as a sequence of bytes). Conversely, its support for a local context of variables and structured programming constructs saves Rupicola from having to grapple with issues such as register allocation or instruction selection.

4.3 Rupicola's source language(s): Coq EDSLs

There is no single, well-define "input language" for Rupicola. Out of the box the compiler supports a number of patterns, and Rupicola's standard library provides support for many more, but extensibility is a key feature: we want users to be able to plug in new transformations, and support both new constructs and new ways to compile already-supported constructs.

In other words, Rupicola's input language is flexible, and how wide a semantic gap to cross as part of the translation done with Rupicola is left to the user's appreciation: users may start from relatively high-level specifications and use a complex set of compilation lemmas, or start from lower-level (yet still purely functional) code but use simpler compilation lemmas. In the upstr example above, for example, users can compile directly from the original upstr with no changes, leaving all complexity to compilation lemmas and rewrite rules; or they may write upstr' as we did but leave the translation from upchar to upchar' to a rewrite rule; or they may plug upchar' directly into upstr', at the cost of making the proof of upstr' against upstr a touch more complex; or they may write an even lower version of the code by appealing to a lower-level loop primitive (nd_ranged_for_all stands for nondependent for loop on all elements of a numeric range — a typical for loop):

```
Definition upstr' (s: list byte) :=
  let/n s := nd_ranged_for_all
               0 (Z.of_nat (length s))
               (fun s idx ⇒
                  let/n b := ListArray.get s idx in
                  let/n b := toupper' b in
                  let/n s := ListArray.put s idx b in
                  s) s in
  s.
```

In our experience, starting from lower-level code (not necessarily as low-level as nd_ranged_for_all above) is almost-always a better approach: it works best to use Rupicola to introduce effects that are unpleasant to encode in the source such as mutation and to handle low-level concerns such as manual memory management, while relying on other high-level techniques to lower abstract specifications into inputs suitable for compilation with Rupicola (the point of this division is to completely separate technicalities of the low-level imperative programming language from proofs of performance optimizations and implementation tricks).

With this layered methodology, which Rupicola shares with languages and frameworks like Low* [52], Fiat [11, 9], and VST [2], users have complete flexibility on how to develop the low-level Gallina code and connect it to high-level specs, all in shallowly embedded style and without worrying about Bedrock2; then, as a completely separate step, the can use Rupicola to jump the verification gap from shallowly embedded Gallina to deeply embedded Bedrock2, with end-to-end proofs.

An example may help make things more concrete, so let us borrow one from Fiat-Crypto [15], where some of my colleagues have been using Rupicola to derive implementations of cryptographic protocols. The highest-level specifications for that code fit in just a few lines (this code was written by Andres Erbsen):

```
Definition poly1305 (p:=2^130-5)
    (k: list byte) (m: list byte): list byte :=
  let r := Z.land (le_combine (firstn 16 k))
                 0x0ffffffc0ffffffc0ffffffc0fffffff in
  let t := fold_left (fun a n ⇒
    (a + le_combine(n ++ [x01])) * r mod p)
    (chunk 16 m) 0 in
  le_split 16 (t + le_combine (skipn 16 k)).
```

Compiled as-is, this code would be very slow, and there are many transformations that we may wish to perform as part of compiling it to Bedrock; a small sample follows:

1. We want to represent lists of bytes as arrays; this is easy to do using a compilation lemma.
2. The code creates new lists from the first and last 16 bytes of k; instead, we would like to take slices of the original array.
3. The computation of r is phrased by converting k into a number and bitwise **and**-ing it with a large constant. A direct implementation of this would require unbounded

integers, which we do not want to introduce here; instead, the constant can be chunked up into bytes and the and operation can be performed over the bytes of the original k.

4. The loop over chunk 16 m performs modular arithmetic on large integers, which we may want to convert to use an optimized implementation.

5. The call to le_combine to construct a number from n and \x01 implicitly allocates memory; ideally, this should be done on the stack

Trying to cram all these transformations into compilation lemmas has two disadvantages. First, the compiler becomes hard to reason about (too much happens for users to comfortably predict characteristics of the output code - tis is one mistake that I made in the design of the Fiat to Facade compiler). Second, the whole compilation process becomes more brittle, because of the amount of reasoning that is performed inside the compiler.

Instead, a different process is much preferable: first write a lower-level version of this code, still in Gallina[10], that is much more explicit about how execution is expected to proceed, and then compile that lowered program using Rupicola. In the example below, the functions used are chosen to all have a non-ambiguous translation to low-level programs.

```
Definition poly1305_impl (k: array_t byte) (msg: array_t byte)
                         (output: Z): array_t byte :=
  let/n (f16, l16) := array_split_at 16 k in
  let/n scratch := buf_make byte felem_size in
  let/n scratch := buf_append scratch f16 in
  let/n (scratch, padding) := buf_split scratch in
  let/n scratch := List.map (fun '(w1, w2) ⇒ byte.and w1 w2)
    (combine scratch [xff;xff;xff;x0f;xfc;xff;xff;x0f;
                      xfc;xff;xff;x0f;xfc;xff;xff;x0f]) in
  let/n scratch := buf_unsplit scratch padding in
  let/n scratch := buf_push scratch x00 in
  let/n scratch := bytes_as_felem_inplace scratch in
  let/n output := felem_init_zero in
  let/n output := array_fold_chunked msg 16
    (fun idx output ck ⇒
       let/n nscratch := buf_make byte felem_size in
       let/n nscratch := buf_append nscratch ck in
       let/n nscratch := buf_push nscratch x01 in
       let/n nscratch := bytes_as_felem_inplace nscratch in
       let/n output := felem_add output nscratch in
       let/n output := felem_mul output scratch in
       output)
    output in
  let/n output :=
    uint128_add (felem_as_uint128 output) (bytes_as_uint128 l16) in
  let/n output := uint128_as_bytes output in
  let/n k := array_unsplit f16 l16 in
  output.
```

Thankfully, connecting these two version of the code is not hard; this is what makes Rupicola's approach viable. Because the code is pure, all the bindings can be unfolded, all the custom functions reduced to primitives that manipulate lists, etc. — all in all, the proof in Coq that relates these two versions is a matter of 5 to 15 lines, depending on how much one cares to automate it.

4.4 The Anatomy of a Rupicola Lemma

Rupicola has two relational compilers: one for expressions and one for statements, each based on the corresponding Bedrock2 judgment. Accordingly, Rupicola extension lemmas are of mainly two kinds.

4.4.1 Expression lemmas

Bedrock2's judgment for expressions, which I write `DEXPR` in Rupicola, is defined such that `DEXPR m l e w` means "expression `e` reduces to machine word `w` when run with memory `m` and locals `l`." Rupicola lemmas for expressions relate `DEXPR` judgments, optionally with preconditions. A typical lemma is thus:

```
Lemma expr_compile_Z_shiftr m l z1 z2 e1 e2:
  0 ≤ z1 < 2^width →
  0 ≤ z2 < width →
  DEXPR m l e1 (of_Z z1) →
  DEXPR m l e2 (of_Z z2) →
  DEXPR m l (expr.op sru e1 e2) (of_Z (Z.shiftr z1 z2)).
```

[10]This version, while more complex, is in fact is arguably closer to the original specification in section 2.5.1 of RFC7539 (<https://datatracker.ietf.org/doc/html/rfc7539#section-2.5.1>) than the more concise specification above.

The function of_Z truncates an unbounded integer. The first two premises restrict the application of the lemma to certain inputs; in this case a right shift on Z (unbounded integers) is expressible as a word operation only if its operand is representable as a machine word and if the shift amount does not exceed the word width (shifting to the left commutes with truncating, but shifting to the right does not, so we must ensure that the upper bits of the original operand are zero). The next two premises are two compilation subgoals for the operand of the shift and the shift amount.

Of course, not every operation maps directly to a Bedrock2 operator; for example:

```
Lemma expr_compile_Z_lnot m l z1 e1:
  DEXPR m l e1 (of_Z z1) →
  DEXPR m l (expr.op xor e1 (expr.literal (-1))) (of_Z (Z.lnot z1)).
```

In these two examples m and l are not used; for l there is really a single lemma (reading a variable from the context), and for m there are many lemmas, with separation logic preconditions asserting the presence of a specific structure at a given address (Bedrock2 expressions include pointer dereferences).

Maps of locals l use a custom notation to improve readability: `#{ ...m; k₁ ⇒ v1; k₂ ⇒ v₂ }#` is short for ``map.put (map.put m k₁ v₁) k₂ v₂.

4.4.2 Statement lemmas

Bedrock2's statement judgment is what Rupicola spends most of its time wrangling with. In Rupicola we write it as a Hoare triple {{ $t; m; l; \sigma$ }} c {{ $P\ p$ }}. In the precondition t is the trace accumulated up to this point in the program, m the memory, l the locals , and σ the environment of functions that the program may call; in the postcondition P is a predicate and p is a Gallina value (the source program); and c is the Bedrock2 program being derived (always an evar). The judgment states that running c with the given starting state (precondition) leads to a final state verifying Pp (here P is partially applied; the result is a predicate on trace, memory, and locals). As an example, here is a statement lemma about translating a replacement in a dependently typed (length-indexed) vector of bytes into a pointer assignment:

```
Lemma compile_vectorarray_put_byte {n} [t m l σ]
      (a: Vector.t byte n) (i: Fin.t n) (b: byte)
      {T} a_var a_ptr (k: _ → T) I B K pred r :
```

```
map.get l a_var = Some a_ptr →
(bytes_vector a_ptr a ⋆ r) m →

DEXPR m l I (of_Fin i) →
DEXPR m l B (of_byte b) →

(forall m',
   let a' := Vector.replace a i b in
   (bytes_vector a_ptr a' ⋆ r) m' →
   {{ t; m'; l; σ }} K {{ pred (k a') }}) →

{{ t; m; l; σ }}
  seq (store access_size.one (expr.op add a_var I) B) K
{{ pred (nlet [a_var] (Vector.replace a i b) k) }}.
```

The lemma has 5 premises. Variables `a`, `i`, and `b` are the arguments to the operation being compiled (seen in `a' := Vector.put a i b` and at the bottom as the argument to `nlet`). The first premise indicates that a pointer `a_ptr` may be found in local variable `var`. The second premise states that memory `m` contains the vector `a` at address `a_ptr`, alongside some separate memory `r`. The third and fourth premises are expression-compilation subgoals for expressions computing the values of `i` and `b` (by convention in this lemma I write deeply embedded terms in uppercase). The final premise is a statement-compilation subgoal: it asserts that a program `K` can be found to implement the remainder `k a'` of the original computation, assuming a modified memory containing the updated vector and the untouched separate memory `r`.

The curious pattern of including a continuation in the compilation lemma allows us precise control on the shape of the next statement-compilation goal. This is in our experience more convenient than using a generic sequencing lemma, in part due to the fact that Bedrock2's predicates are asymmetric (the precondition is not a predicate but a collection of values that are universally quantified over in the Coq context of the proof).

Also worthy of note is the way the continuation is invoked: the `nlet` form includes not only a value and a continuation (the usual encoding of (`let var := val in body`) as (`(fun var ⇒ body) val`)) but also a list of names (`[a_var]` in this case) that are provided by the user to describe which objects are intended to be mutated[11].

```
Definition nlet {A T} (vars: list string) (a: A) (body: A → T) : T :=
  let x := a in body x.
```

[11]A variant of this lemma exists that ignores the variable and instead searches for any pointer to `a`, but

This wrapper has three purposes:

1. It helps direct the compiler: users introduce `nlet` bindings to direct the compiler to overwrite or mutate existing bindings, or to create new ones.
2. It prevents overly aggressive reduction: many Coq tactics tend to inline let bindings (ζ-reduction) as a side-effect of performing some other useful task.
3. It materializes the body of the let binding as a function — there is no other way to write a Coq lemma that matches a let bindings parametrically on its body without relying on higher order unification.

The vars part of `nlet` are automatically captured by a Coq notation [46], so the experience of authors of source programs is the same as writing regular let bindings, only with additional meaning given to the choice of names[12]:

```
Check let/n x := 1 in x + x.
```

```
nlet ["x"] 1 (fun x : nat ⇒ x + x)
     : nat
```

In fact, for maximal generality, Rupicola compilation lemmas are usually written using a *dependently typed* version of `nlet`, named `nlet_eq`:

```
Definition nlet_eq {A} {P: forall a: A, Type}
           (vars: list string) (a: A)
           (body: forall a' (Heq: a' = a), P a') : P a :=
  let x := a in body a eq_refl.
```

The reason this variant is needed is that, in Coq, the forms `let x: T := a in body` and `(fun x: T ⇒ body) a` are not equivalent[13]. The former is transparent, in the sense that *body* may access depend on the value of a for typechecking purposes, whereas the later is opaque, in that it may only depend on the type *T* of x:

making the name explicit makes compilation more predictable: without it Rupicola might pick the wrong object to mutate if there were two indistinguishable objects present in memory at a given point in a compilation run.

[12] In other words, the `nlet` form itself preserves α-equivalence at the semantic level as it discards its var argument, but the compiler takes it into account.

[13] This fact was first pointed to me by Jasper Hugunin in the thread *Unfold a constant in Program constraints* on Coq's Zulip channel.

```
Compute let n: nat := 3 in Vector.hd (Vector.const true n).
```

```
= true
: (fun (n : nat) (_ : Vector.t bool (S n)) ⇒ bool) 2 (Vector.const true 3)
```

```
Compute (fun n: nat ⇒ Vector.hd (Vector.const true n)) 3.
```

```
In environment
n : nat
The term "Vector.const true n" has type "Vector.t bool n"
while it is expected to have type "Vector.t bool (S ?n)".
```

The first program computes the head of a constant vector containing 3 elements; this is a well-typed operation, because 3 is `S 2`, and the type of `Vector.hd` requires a vector whose length matches `S _`:

```
Check @Vector.hd.
```

```
@Vector.hd
     : forall (A : Type) (n : nat), Vector.t A (S n) → A
```

The second program fails, because the body of the function does not have access to the value of `n`: the function itself must typecheck in all circumstances. Trying to write the program with `nlet` fails in the same way, but it succeeds with `nlet_eq`, at the cost of an explicit cast:

```
Compute let/n n := 3 in
        Vector.hd (Vector.const true n).
```

```
In environment
n : nat
The term "Vector.const true n" has type "Vector.t bool n"
while it is expected to have type "Vector.t bool (S ?n)".
```

```
Compute let/n n eq:Heq := 3 in
        Vector.hd (rew Heq in (Vector.const true n)).
```

```
= true
: (fun _ : nat ⇒ bool) 3
```

Part of Rupicola's default compiler automation transforms `nlet` into `nlet_eq` before looking for lemmas to apply — this allows authors to write phrase compilation lemmas in terms of `nlet_eq` and have them apply to both `nlet_eq` and plain `nlet` goals (without this there would have two be two lemmas per code pattern, one for `nlet` bindings and one for `nlet_eq` bindings).

4.5 Relational compilation in Rupicola

Rupicola mostly follows the structure laid out in the later parts of chapter 3, but two aspects are particularly interesting and worthy of discussion: effects and control flow.

4.5.1 Compiling effectful programs

A key tool to achieve excellent performance for extracted program is leveraging the target language's native effects. In Rupicola, effects are classified into two categories: intensional and extensional.

4.5.1.1 Intensional effects

Intensional effects are not explicitly encoded in the source (they do not appear in type signatures). Instead, they are introduced by special-casing certain code patterns through compiler extensions.

State and certain aspects of allocation are handled this way in Rupicola. For state, in particular, we do not typically use an explicit encoding: instead, we add lemmas to map e.g. list accesses to pointer dereferences, or pure replacements in a list to pointer assignments. Allocation of short-lived objects on the stack is handled similarly; I discuss it in a case study in section 5.1.2.1.

In general, intensional effects are either inferred or introduced explicitly using semantically transparent annotations on source programs. Every let-binding in functional models fed to Rupicola, for example, is annotated with the name of the variable it binds, allowing the compiler to make decisions about mutating existing variables and objects or creating new ones based on the user's choice of names (in general, Rupicola expects input programs to be sequences of let-bindings, one per desired assignment in the target language). Similarly, to indicate that a let-binding should result in a copy instead of a mutation, a user might wrap the value being bound in a call to a `copy` function of type $\forall\, \alpha . \alpha \to \alpha$. Finally, while in simple cases data-structure mappings can be inferred automatically, in complex cases the user can control memory layout explicitly by using modules that transparently wrap underlying functional types (for example, the `ListArray` module reexposes list operations in a semantically transparent way that Rupicola can still use to drive compilation).

With this lightweight approach to intensional effects, and especially mutation, compiled programs can make full use of low-level state while source programs remain easy to reason about, with no explicit heap at the source level. This is a key advantage of Rupicola's intensional encoding of effects: it essentially does not impede verification efforts. When proving a functional model against a higher-level specification, annotations can simply be unfolded away: Rupicola's name-carrying let-bindings unfold to regular let-bindings, functions like `copy` above simply disappear, and modules wrapping standard types unfold to reveal them.

4.5.1.2 Extensional effects

Extensional effects, in contrast, are introduced using explicit monadic encodings. This is how Rupicola handles nondeterminism and I/O, but the methodology generalizes.

Specifically, Rupicola's encoding of postconditions is designed to be monad-agnostic, in the sense that lemmas about nonmonadic terms apply regardless of the monad that the program is using. This consolidation is possible because the postcondition of a Rupicola compilation goal is split between a term being compiled and a predicate on that term. The choice of monad changes the type of the predicate but not the (head of the) term, when that term starts with a pure computation (such a term would look like `let x := a in k x`, where `k` might have a monadic type). Writing lemmas about nonmonadic computations parametrically on the predicate and the current continuation guarantees that they are applicable regardless of the ambient monad.

Lemmas about monadic computations, on the other hand, are by definition not monad-agnostic: they recognize terms of the form `bind ma k`, refine the current low-level program with an implementation of `ma`, and produce a goal for the continuation with a term of the form `k a` for some `a`. For this last step to be possible, the predicate being compiled need to obey certain properties: in short, given `P: M A → state → Prop` and a term `bind ma k`, we need to find a relation between `P (bind A k) st` and `P (k a) st` for all `st` and for some (potentially universally quantified) value `a`. We guarantee alignment by using a monad-specific *lift* when compiling monadic programs, so that the postcondition always has shape `lift P (bind ma k)`.

The following two examples illustrate this pattern:

For the nondeterminism monad, we can encode nondeterministic computations returning a value of type `A` as `A → Prop` (for example, a list of `n` unspecified natural bytes is represented as `(fun bs: list byte ⇒ length bs = n)`). Then, we just need to require predicates to be lifted using the function $P \mapsto \lambda$ ma *st*.$\exists\ a,$ ma $a \wedge P\ a$ *st*, which is such that {{ $t; m; l; \sigma$ }} c {{ lift P (bind ma k) }} is implied for all a by ma $a \wedge$ {{ $t; m; l; \sigma$ }} c {{ lift P ($k\ a$) }} (this is similar to what happens with nonmonadic bindings presented in section 4.4, but the value is constrained by the computation ma).

As an example, the following lemma exposes stack allocation as a nondeterministic computation. The lemma assumes that an arbitrary predicate holds about the memory m, and runs the continuation k with that same memory augmented with an array of undetermined values. While the initial goal (bottom) contains a monadic bind, the continuation goal does not; instead, bs is a universally quantified list of bytes, only subject to the contrainst length bs = nbytes. Note also how the predicate pred changes, to allow the stack-allocated memory to be released after the execution of k completes.

```
From Rupicola Require Import NonDeterminism.

Definition stack_alloc (nbytes: nat) : ND.M (list byte) :=
  (fun bs ⇒ length bs = nbytes).

Notation lift := ndspec_k.
Lemma compile_stack_alloc [t m l σ] (sz: nat) :
  forall B (pred: B → predicate) (k: list byte → ND.M B) K r var,

    r m →
    sz mod (Memory.bytes_per_word width) = 0 →

    (forall ptr bs m',
        length bs = sz →
        (nbytes sz ptr bs ⋆ r) m' →

        let pred g t' m'' l' :=
            ∃ r' bs', (nbytes sz ptr bs' ⋆ r') m'' ∧
                      forall m0, r' m0 → pred g t' m0 l' in
        {{ t; m'; #{ … l; var ⇒ ptr }#; σ }}
          K
        {{ lift pred (k bs) }}) →

    {{ t; m; l; σ }}
      cmd.stackalloc var sz K
    {{ lift pred (mbindn [var] (stack_alloc sz) k) }}.
```

For the writer monad, we can encode a computation as a pair of a value and some accumulated output, and require predicates to be lifted using the function $(o, P) \mapsto \lambda$ ma *st*.P (fst ma) (o ++ snd ma) *st*. Parameter o of the lift accumulates previous output, allowing us to compile monadic binds by accumulating their output into that parameter while reducing the source term. Here the relation is that $\{\{\ t; m; l; \sigma\ \}\}\ c\ \{\{\ \texttt{lift}\ o\ P\ (\mathsf{bind}\ \mathtt{ma}\ k)\ \}\}$ is implied by $\{\{\ t; m; l; \sigma\ \}\}\ c\ \{\{\ \texttt{lift}\ (o +\!\!+\ \mathsf{snd}\ \mathtt{ma})\ P\ (k\ (\mathsf{fst}\ \mathtt{ma}))\ \}\}$

Thanks to this approach, only control-flow lemmas and lemmas that implement monadic computations need to be programmed.

4.5.2 Predicate inference for conditionals and loops

Compiling loops and conditionals poses specific challenges in Rupicola. To understand why, recall that the compilation process often needs to consult the current precondition, which captures the state reached after symbolically executing the already-derived prefix of the output program. This means that Rupicola needs careful control on the shape of the preconditions that get derived as compilation progresses; in other words, Rupicola needs to *infer* invariants and predicates at control-flow join points — a well-known challenge in automated verification (for readers familiar with predicate-transformer semantics, Box 1 gives a different intuition about the difficulty of compiling loops and conditionals in Rupicola).

Thankfully, the specifications for the loops and conditionals that Rupicola compiles are pure functional programs, and hence may appear directly in invariants. As a result, loop invariants can be inferred automatically and predictably, by capturing strongest postconditions in terms of partial executions of the functional model that defines a loop.

I start by giving the outline of one loop-compilation lemma, before explaining how loop predicates are computed.

4.5.2.1 The issue with branches

For straightline programs, the postcondition used to drive the compiler is given by the user in the form of a distinguished value (the source program) and a predicate relating that value to the final state of the low-level program (in terms of predicates about local variables and about the heap). The predicate is not inspected until the very last step, while the source program is progressively consumed, moving bindings into the context of the derivation. Each new binding yields exactly one new compilation goal, for the continuation of the program. For programs with conditionals, loops, and other forms of control flow, we have

Box 1 — *Rupicola as a Predicate-Transformer Calculus*

To understand the challenges with nonlinear control flow in Rupicola, it may help to think of compilation lemmas as specialized rules plugged into an extensible predicate transformer.

A Rupicola compilation run begins with a proof context describing a symbolic state (local variables using a map of strings to machine words and the heap using a separation-logic predicate). With each application of a compilation lemma, we (1) make progress in the derivation of the low-level program, (2) record the effect of running that newly added part of the low-level program on the symbolic state, and (3) simplify the source program, moving its head binding into the context of the current proof.

Part (1) is simply generating a program, but parts (2) and (3) describe a postcondition calculation: specifically, part (2) is one step in the computation of a postcondition capturing the semantics of the partial evaluation of the target program up to the current point in the derivation, and part (3) takes a similar step in the evaluation of the source program.

And, as always with predicate-transformer semantics, conditionals and loops require special care: while it *is* possible to automatically derive a strongest postcondition for conditionals *and* for loops (since loops are guaranteed to terminate), the results of that computation would not be in a shape compatible with the rest of the compilation process. (This is another way in which Rupicola codifies and automates reasoning patterns that are otherwise performed manually: when verifying a handwritten Bedrock2 program, users will commonly develop custom lemmas to relate the execution of a low-level fragment to a convenient functional model, and these lemmas have a shape very similar to Rupicola compilation lemmas. Rupicola's insight is that with just a bit more structure, we can use these lemmas to derive the low-level code directly from the functional model.)

multiple compilation goals corresponding to multiple program paths (one per branch of a conditional, or one for the body of a loop, plus one for the continuation of the program). These goals do not have the same postcondition as the original program[14], and depending on how we state these postconditions, we may not be able to continue compiling.

This concern applies to all forms of branching: introducing a new join point in the control-flow graph of the output program requires stating or inferring a predicate that characterizes its locals and memory at the join point, and that predicate needs to be in a shape that can be exploited by the rest of the compilation process.

To illustrate the issue, let us start with the simpler case of finding an invariant that holds after a conditional. Suppose that we are compiling code that writes value `x` to a memory cell at address `p` conditionally on a test `t` and returns a Boolean indicating whether a write has happened (in code: `let (r, c) := (if t then (true, put c x) else (false, c)) in k c`, a trivial compare-and-swap with `k` standing for the program's continuation), with locals `{"c": p}` and a memory predicate `cell p c` (stating that cell `c` is in a block of memory at address `p`). To compute the next symbolic state (the precondition used to compile `k`), we might naively attempt to compile both branches and then use a naive merge of their strongest postconditions. The result, unfortunately, is a new predicate `(t ∧ cell p (put c x)) ∨ (¬t ∧ cell p c)` that is incomprehensible to later compilation steps: code in `k` will be looking for a single `cell` predicate mentioning the new value of `c` (`if t then … else …`), not a disjunction.

The situation is even worse with loops: compiling the body of a loop requires a concrete precondition (a symbolic state that we can inspect), and since that precondition needs to have the same shape as the postcondition of the body (both of these are instances of the loop invariant), we cannot even consider delaying the computation of the postcondition.

4.5.2.2 Rupicola's solution

The solution I implemented in Rupicola is based on a relatively simple heuristic. It takes advantage of the chance to impose rules on exactly which functional programs Rupicola will accept, such that a prepass of proved program transformation may be needed to enable compilation, but in return compilation is very predictable. The algorithm is as follows:

[14]An exception is when the postcondition is for a conditional that appears in tail position in the program, in which case things are essentially the same as for a linear program.

1. Identify targets of the control-flow construct (loop or conditional) based on the names in the corresponding bindings. In the compare-and-swap example above, this would be two variables, `"r"` and `"c"`.
2. For each target, determine whether it is a scalar or a pointer by inspecting the current locals and memory predicate. In the CAS example, we would determine that `"r"` is a scalar and `"c"` is a pointer: `"r"` because we do not find a binding for it in the map of locals, and `"c"` because the binding we find for it (`"c": p`) is to a pointer (`p` appears in the separation-logic predicate `cell p c`).
3. For each scalar, generalize over the corresponding binding in the locals. For each pointer, generalize over the corresponding entry in the predicate describing the memory. For CAS, we build a new map of locals `{"c": p, "r": _}` and a new memory predicate `cell p _`.
4. Close over the results. For CAS, we obtain the predicate (`fun '(r, c) l m ⇒ l = {"c": p, "r": r} ∧ (cell p c) m`).

The resulting predicate is parameterized on the source-level values of the variables being created or mutated: to obtain a plain predicate, we need to plug them in. For forward edges (conditionals) they are exactly the source program being compiled, so for CAS we obtain (`fun l m ⇒ let (r, c) := (if t then (true, put c x) else (false, c)) in (l = {"c": p, "r": r} ∧ (cell p c) m)`).

Loops are trickier to deal with: for backwards edges (loop invariants), we need a source-level characterization of *partial* progress through the loop to close the parameterized predicate derived using the algorithm above. In other words, when stating a loop-compilation lemma, we need to give not just the precondition of the continuation, which refers to the results of iteration, but also the pre- and postconditions of the loop body, which refer to partially processed inputs.

In traditional verification, this would done by asking users to supply manually crafted invariants. In Rupicola, however, we have an easier way: since we are only concerned with compilation of functional programs using constructs from a curated menu (we have one compilation lemma per type of loop), we can build the “symbolic” variable values corresponding to partially processed inputs simply by running the functional program for a reduced number of iterations!

In general, these partial-progress characterizations are very easy to express: for iterators on lists, they correspond to iterating on a prefix of the list while leaving the rest untouched; for iterators on ranges of numbers, they correspond to iterating on the first part of the range.

For example, suppose we are compiling the loop `let c := Nat.iter 10 incr c in k c` (where `incr` increments the content of a cell, and `Nat.iter n` composes a function with itself `n` times). We obtain a general invariant (`fun i l m ⇒ let c := Nat.iter i incr c in l = {"c": p} ∧ (cell p c) m`), where the value of `c` is derived from the number of already-completed iterations `i`.

This process works without extensions for all examples presented in this thesis. There are a few cases that it does not handle, but fewer than it may seem (for example, arithmetic on pointers may appear not to be inferred correctly, but arithmetic on pointers is in fact not expressible as such in the source language). Most of them stem from simplifications in step 2: first, as written, it will always attempt to swap the contents of objects and never swap pointers (for example, `let (x, y) := if … then (y, x) else (x, y) in …` will swap the contents of `x` and `y` word by word, rather than reassigning the corresponding pointers); second, it will always assume that new bindings refer to new scalars (no conditional allocations). The first issue can be solved by adding semantically transparent annotations in the source; so can the second, but in practice it is seldom a concern for the programs we consider.

4.5.2.3 A note on loop invariants

Conveniently, this approach solves a second concern, specific to loops: while there may be properties that we want to prove across loop iterations, it would be a significant issue if users had to write invariants mentioning low-level states to supplement their purely functional code (it would break Rupicola's promise that translation to deeply embedded imperative programs can be automated and decoupled from writing the purely functional models). For example, we may need to prove that a particular value is in bounds (in the `incr` example above, maybe the contents of the cell are then used to index into an array). Usually, these sorts of properties are expected to be plugged in by users, but we do not want users to have to state properties in terms of the memory or the locals of the program.

In practice properties of interest fall into two categories, and these two are handled differently:

4.5.2.3.1 Structural properties Properties inherent to the choice of representation of a value (we call them *structural*) are encoded in separation-logic predicates. This is the case for properties like the length of an object not changing when it is mutated, for example. (this is a concern when running a loop over an array: how do we know whether a later array access is valid, if an earlier iteration of the loop might have grown or shrunk the array?) Concretely, in our original uppercasing, we chose a separation-logic predicate that captured the length of the string in addition to its contents. Structural properties are automatically captured by our loop-invariant inference, so encoding properties structurally is almost always beneficial.

4.5.2.3.2 Incidental properties Properties specific to a particular algorithm or program (we call them *incidental*) are proven at the source level and recovered during compilation using hints. For example, if in addition to incrementing a cell our loop also accessed an array at the index corresponding to the value of the cell (`arr[*p]`), we would want to prove that after each iteration, the value in the cell is still within the bounds of the array.

With our approach, rather than encoding them as low-level loop invariants, users prove structural properties directly at the source level, by proving theorems about partial executions of their loops (iterations over part of a list, or a range of numbers). For example, a user may prove that for all `i`, `get (iter i incr c)` equals `get c + i`. Plugging this as a compilation hint would then allow a linear solver to prove side conditions like `0 ≤ get (iter i incr c) ≤ length arr` from preconditions about `c` and `length arr`.

Early versions of Rupicola did not perform loop invariant inference, treated all properties as incidental, and pushed all reasoning to compilation time. As a result, applying loop lemmas required stating complex invariants and proving them by reasoning directly on low-level Bedrock2 state. This led to slow, brittle derivations that performed complex reasoning and tended to be hard to automate. In contrast, by eliminating all ghost state and enforcing a specific shape of invariants, the new loop lemma lend themselves nicely to automated derivation.

A good example of the benefit of this approach is given in the discussion of the Montgomery ladder function in section 5.1.5.1.

4.6 Rupicola's Architecture

Rupicola is divided into a minimal core and a collection of orthogonal extensions that enables users to grow its input language. Not all programs use all these extensions: users are able to mix and match, and in fact throughout Rupicola's development a common pattern has been to implement support for new constructs as part of the development of an application and later to move these parts into Rupicola's standard library.

4.6.1 The core

Rupicola's core is composed of definitions, notations, forward-reasoning tactics, and supporting architecture; together, they are enough to state program-compilation goals and run the compiler but not to derive any concrete programs: almost all features of Rupicola are implemented as extensions of this minimal core.

4.6.1.1 Core definitions and notations

Rupicola's compilation relations are exactly the semantic judgments of Bedrock2 for expressions and statements, applied to postconditions with a distinguished argument indicating which program is being compiled. We write them as `DEXPR m l e w` and `<{ Trace := t; Memory := m; Locals := l; Functions := σ }> c <{ P p }>` respectively, with `m` a memory, `l` a map of locals, `e` an expression, `w` a word, `σ` an environment of functions, `c` a command, `P` a predicate, and `p` a Gallina value (the `{{ … }}` notation that we encountered previously for statements is an abbreviation of the statement judgment notation when there is no ambiguity).

Rupicola has notations for program specifications and compilation goals, which we have briefly encountered previously; specifically, the notation for specifications attaches a pre- and post-condition pair to a name:

```
Instance spec : spec_of name :=
  fnspec! name args / ghost_args, {
    requires tr mem := precondition;
    ensures tr' mem' rets := postcondition
  }.
```

It desugars to a forall-quantified statement (here `[…name]` is a shorthand notation used to indicate that we are constructing a list from the names in *args*):

```
Instance spec : spec_of name := fun σ ⇒
  forall args ghost_args, forall tr mem,
    precondition →
    WeakestPrecondition.call
      σ name tr mem [...args] (fun tr' mem' rets ⇒
         postcondition).
```

The notation for compilation is intended to be used in tandem with Coq's `Derive` statement, which provides convenient syntactic sugar to generate a term its proof in tandem.:

```
Derive impl SuchThat
  (defn! name (args) ~> rets { impl },
   implements spec using fns)
As proof.
```

It desugars to a theorem stating that a piece of syntax *impl* verifies the specification above (it finds that specification through typeclass resolution, based on the function's name), roughly equivalent to the following. The notation ...*fnspec* → stands for a sequence of implications, one per function in *fns*, each asserting the specification of that function. The premise `__rupicola_program_marker` *spec* is vacuous; it simply helps Rupicola pick out the program being compiled from the postcondition.

```
Derive body SuchThat
  (__rupicola_program_marker spec →
   forall σ, ...fnspec σ →
       spec ((name, ([args], [rets], body)) :: σ))
As proof.
```

Finally, Rupicola's core definitions include a type class that identifies which binding constructs are Rupicola binders; this is used when inspecting a program to determine whether it is fully compiled (Rupicola assumes that input program are written as a sequence of let bindings, each of which gets compiled to an individual assignment or mutation in the target language[15]):

```
Inductive RupicolaBindingInfo :=
| RupicolaBinding (rb_type: Type) (rb_names: list string)
```

[15]Compiler builders are free to plug rewrites into the compiler that new introduce bindings, e.g. by splitting a larger term into a sequence of subterm calculations, though in our experience manually introducing bindings leads to more reliable compilation, since Rupicola's mixed embedding of binders allows it to straightforwardly determine which operations should introduce mutation and which ones should not.

```
| NotARupicolaBinding.
Class IsRupicolaBinding {T} (t: T) :=
  is_rupicola_binding: RupicolaBindingInfo.
```

Instances of this typeclass are derived dynamically, using a tactic and defaulting to `NotARupicolaBinding`. Here is an example from Rupicola's standard library:

```
Hint Extern 2 (IsRupicolaBinding (nlet (A := ?A) ?vars _ _)) ⇒
  exact (RupicolaBinding A vars) : typeclass_instances.
```

This allows users to register hints to extend the set of patterns that are considered compilation targets. The compiler uses these hints to decide whether it should attempt to invoke additional compilation lemmas or whether it should try to unify the pre- and post-condition of the program.

The first implementation of Rupicola did not have this type class: Rupicola just tried to apply lemmas as long as they were available, and then tried to unify the pre- and post-conditions of the program. The problem with that approach is that makes debugging and incremental compiler construction significantly more difficult, since the user is presented with a unification failure instead of a helpful message highlighting which construct is unsupported by the compiler. Incremental compiler construction is discussed in more detail in section 6.2.

4.6.1.2 Compilation tactics

The rest of Rupicola's core defines tactics for driving the proof-search process. Rupicola's compiler is a single Coq tactic that runs the following steps in a loop until either the goal is solved or no more progress is possible:

1. Goal clean-up and context management: introduce quantified variables, prune stale hypotheses, substitute variable equalities, and perform other user-specified cleanups.
2. Compilation: find a lemma that applies to the current goal from the current database of lemmas and apply it. This is done in forward-reasoning style; the details are described in the following section.
3. Automatic side-condition resolution: use generic and user-specified solvers to discharge side conditions arising from the use of compilation lemmas.

Additionally, a set-up phase prepares the goal for compilation, translating the `defn!` goal into a compilation goal stated as a Hoare triple. This translation phase also applies a few compilation passes on the generated Bedrock2 AST, to make Rupicola's output more readable: a transformation that remove Bedrock2 `skip`s (no-ops), and another that remove self-assignments (`a = a`). It may be surprising that these transformations are applied as part of the *setup* phase of the compiler, before code is actually generated, but that is justified by the structure of Rupicola's compilation goals. Indeed, starting with a goal of the form `{{ t; m; l; σ }} ?c {{ P p }}`, applying a transformation `T` to resulting program `?c` can be done by ensuring that `?c` be in the shape `T ?c'`, for some other program `?c'`. The correctness of `T` is captured by a compilation lemma that states that if `{{ t; m; l; σ }} ?c' {{ post }}` holds, then so does `{{ t; m; l; σ }} T ?c' {{ post }}`. Applying such a lemma as the very first step of compilation unifies ``?c`` with `T ?c'`, and the result after finishing to compile is a program wrapped in the transformation `T`.

4.6.1.3 Extension points

Rupicola's strength is its extensibility, so we need users to be able to straightforwardly plug in new lemmas and new reasoning. Rupicola exposes the following extensions points:

`compiler_setup` (`fw`) Used to preprocess compilation goals without bypassing the default setup automation.

`compiler_setup_post` (`fw`) Used to preprocess compilation and bypass part the default setup automation. This is used by monadic programs: the default setup automation refactors the postcondition of the compilation goal as `(P p)`, where `P` is a predicate and `p` a Gallina term, but for monadic programs we need `P` to have a specific shape `lift P p` (section 4.5.1.2). Part of implementing support for a new monad is to add a lemma to this database that refactors the postcondition into this lifted shape (the default automation only runs if none of the lemmas in this database apply).

`compiler_cleanup` (`fw`, `rw`, `u`) Used to preform misc cleanups at each step of the compilation process, before applying compilation lemmas. This is used to simplify expressions (e.g. repeated casts like `Z.to_nat (Z.of_nat …)`), inline functions, plug in optimizations expressible as rewrites, etc.

`compiler_pull_binding` (`fw`) Used just before converting `nlet`s into `nlet_eq`s. This is used to apply transformations that are significantly more complex to state in terms of the dependently-typed `nlet_eq` — typically transformations that reorder bindings, for example to flatten a nested tree of bindings such as `let/n x := (let/n y := 1 in y) in x`.

`compiler` (`fw`) The first of two main compilation entry points. This hook is used to plug in statement compilation lemmas.

`expr_compiler` (`fw`) The second of two main compilation entry points. This hook is used to plug in expression compilation lemmas.

`compiler_side_conditions` (`fw`, `rw`, `u`) Used to solve side conditions that arise from applying lemmas from `compiler` and `expr_compiler`.

`compiler_cleanup_post` (`fw`, `rw`, `u`) Used for aggressive clean ups that precede unification when compilation is complete.

All of these hooks are Coq hint databases; the ones marked `fw` (for `forward`) are only used with `typeclasses eauto`: lemmas added to them with a call to `shelve` make forward progress, and other lemmas added to them are only used if they form a complete path to a solution of the current goal; the ones marked `rw` and `u` are also used for rewriting and unfolding, respectively.

4.6.2 Rupicola's standard library

Beyond the core compilation support, Rupicola's standard distribution includes support for the following features:

4.6.2.1 Arithmetic

Rupicola's expression compiler supports arithmetic on `nat`, `N`, `Z`, `byte`, `bool`, `Fin.t`, and machine words. Not all types support all operations, but new operations and new types are very easy to add (details on extending the expression compiler are given in section 5.1.3).

The expression compiler is mostly straightforward, but one pattern is tricky, and is in fact the only place where Rupicola currently uses backtracking[16]: comparisons on unbounded integers. There are two ways to map a comparisons on integers to a Bedrock2 operations: the `ltu` and `lts` binary operators, which compare their operands are unsigned or signed, respectively. Both of them require proving that the original integers are bounded, but the bounds are different[17]: `ltu` is valid if the integers are in the range $0 \leq \ldots < 2^w$, while `lts` is a valid implementation of `Z.lt` if the integers are in the range $-2^{w-1} \leq \ldots < 2^{w-1}$.

This issue is handled by not shelving the side conditions generated by the `expr_compile_Z_ltb_u` and `expr_compile_Z_ltb_s` lemmas: this way, unless we can immediately prove the required bounds, compilation stop without making potentially incorrect assumptions that would later lead to unsolvable goals. Of course, if a user wants to apply either lemma unconditionally, they are free to do so by registering hints for these lemmas with an additional call to `shelve`.

4.6.2.2 Arrays

Rupicola has five encoding of arrays: unsized list arrays, sized list arrays, vector arrays, buffers, and inline tables.

The first three share a common API: `get`, `put`, and `map`; they are all mapped to flat arrays of immediates (bytes, or native ints), but they are parametric on the type of data being stored in the array, so in addition to arrays of bytes and native ints it is possible to use byte arrays to store values of any type that is representable within a byte, and word arrays for any type whose values fit in a machine word. Concretely, that means that it is possible to have, say, arrays of tagged values, e.g. encoded values with three bits used to store a tag and the rest used to store an immediate value, or arrays of subsets of integers, e.g. an array of numbers less than 7 (this helps when indexing repeatedly, e.g. when the result of one array lookup is used to index into another array: if the first array has values of type `{x | x < length arr2}`, then the bound checks are trivial in the second array).

Buffers are also flat arrays of words under the hood, but they are partially uninitialized: the Gallina model of these values only captures the initialized part, but the separation logic predicate also includes additional unknown (existentially quantified) values:

```
Definition buffer_value (ptr: word)
           (data: list word) (capacity: nat) (m: mem) :=
  ∃ padding: list word,
    sizedlistarray_value AccessWord capacity ptr (data ++ padding) m.
```

[16] In general, backtracking is not desirable; for a discussion of the reasons, see section 6.2.

[17] In contrast, the is a single `eq` operator, and it requires that both of its operands be in the same bounds, but these bounds may be either the signed or the unsigned ones.

The buffer API allows pushing words if there remains uninitialized data; popping words if there remains initialized data; and converting into a regular buffer of bytes using the sized array predicate if the buffer is full.

The inline tables API is similar to the sized array API, but read-only and statically allocated; I discuss it in detail in section 5.1.2.2.

4.6.2.3 Control flow

Out of the box, Rupicola has support for conditionals (if expressions) and for a variety of loops. For conditionals there are two lemmas: one for conditionals in tail position, which do not require predicate inference since the control flow join point after such a conditional is the function exit, and one for conditionals in expression position, which do require predicate inference (I discussed predicate inference in detail section 4.5.2).

For loops, Rupicola has a family of generic lemmas, with support for higher-order iteration patterns is built on top of it. Specifically, the library builds the following definitions:

`foldl_dep`, `foldl` A dependently typed (resp. simply typed) left fold (the body of the fold gets access to a proof that the element is part of the original list), with support for stopping part-way through the traversal (encoded as a stopping condition `stop: A -> bool`).

`ranged_for_break`, `nd_ranged_for_break` Like `foldl_dep`, but specialized to a range of numbers (`nd` stands for nondependent).

`ranged_for`, `nd_ranged_for` Like `ranged_for_break`, but with a flag set by the body of the loop to indicate early exits instead of having a separate stop condition.

`ranged_for_all`, `nd_ranged_for_all` Like `ranged_for`, but without early exits.

`ranged_for_u`, `nd_ranged_for_u`, `ranged_for_all_u`, `nd_ranged_for_all_u`, `ranged_for_s`, `nd_ranged_for_s`, `ranged_f`
Like `ranged_for` and `ranged_for_all`, but specialized to ranges of machine words instead of unbounded integers.

On top of these definitions we build a core loop compilation lemma for the dependently typed version of `ranged_for`, and then derive a collection of more convenient lemmas from it, starting with lemmas that initialize loop variables. Since Bedrock2 does not have a `break` statement, early exits are encoded by having the body set the loop counter to the maximum value.

All definitions past `ranged_for`, as well as `maps` and `folds`, are compiled by rewriting them in terms of `ranged_for` and then applying the `ranged_for` compilation lemma, or using specialized lemmas derived from that lemma and customized to apply to these higher-level iterators. The latter requires a bit of additional development (new compilation lemmas), but it is in general much preferable to the former, since it allows the intermediate state exposed in the body of the loop to be expressed in terms of the higher-level iterator: for example, compiling a `map` by rewriting it to a `ranged_for` gives an intermediate state with a mutated array `l' = ranged_for 0 k (fun k l Heq ⇒ put k (f (get k l))) l`, whereas using a specialized loop lemma gives `l' = map f (firstn k l) ++ (skipn k l)` — both are equivalent, but the former is much easier to work with.

This last point is a special case of the encoding difficulties that come up with loop lemmas: loop lemmas are particularly tricky because they tend to have more complex side-conditions, and these side-conditions need to be phrased in a way that is amenable to automated resolution. As a standalone case study, let us look in more details at the simplest loop compilation lemma: the one for `Nat.iter`.

```
Lemma compile_Nat_iter [t m l σ] n {A} f (a: A) :
  let v := Nat.iter n f a in

  forall B (pred: B → predicate) (loop_pred: nat → A → predicate)
    (k: A → B) K F I i_var vars,

    n < 2 ^ width →
    DEXPR m l I (of_nat n) →
    loop_pred n a t m #{ … l; i_var ⇒ of_nat n }# →

    (forall i st t m l, (* predicate stability *)
       loop_pred i st t m l → map.get l i_var = Some (of_nat i)) →

    (let loop_pred := loop_pred in (* loop body *)
     forall t l m i, i < n →
     let st := Nat.iter (n - S i) f a in
     loop_pred (S i) st t m l →
     {{ t; m; #{ … l; i_var ⇒ of_nat i }#; σ }}
       F
     {{ loop_pred i (f st) }}) →

    (let v := v in (* continuation *)
     forall t l m, loop_pred 0 v t m l →
     {{ t; m; l; σ }} K {{ pred (k v) }}) →

    {{ t; m; l; σ }}
```

```
        cmd.set i_var I;;
        cmd.while (expr.op ltu 0 i_var)
          (cmd.set i_var (expr.op sub i_var 1);; F);; K
    {{ pred (nlet vars v k) }}.
```

This lemma has 6 premises:

1. `n < 2 ^ width` guarantees that the number ot iterations we are performing fits in a machine int.
2. `DEXPR m l I (of_nat n)` is a compilation subgoal: it requires us to compile an expression that reduces to the number of iterations that we want to perform.
3. `loop_pred n a t m #{ … l; i_var => of_nat n }#` requires the loop predicate to hold when the loop starts. It is not invoked with the original locals `l`, but with the result of adding `i_var` to `l`, because the code that we generate initializes a fresh loop counter `i_var` (`cmd.set i_var I`).
4. The `predicate stability` premise ensures that the loop body does not overwrite the loop counter.
5. The `loop body` premise is a compilation goal: it requires us to compile the function `f`. There are a few interesting aspects to highlight. First, we have a seemingly redundant binding `let loop_pred := loop_pred`: adding this binding prevents Coq from inlining `loop_pred` into the postcondition of our compilation goal, which maintains the expected shape of the postcondition (`P p` with `P` a predicate and `p` a Gallina program). Second, we have the assumption `i < n` and the state `st`; note how `st` is fully concrete and defined in terms of the high-level iterator, `Nat.iter` (not in terms of the Bedrock2 implementation of the body of the loop). Third, we have the loop predicate, applied to `S i`, not `i`: this is because the loop body starts by subtracting 1 from the loop counter, so at the beginning of the loop the subtraction has not happened yet. But, fourth, the body of the function itself is invoked with a map of locals that contains `i` for `i_var`, not `S i`, since by the time the Bedrock2 implementation `F` of `f` starts running the loop counter decrement has happened (hence the `i` also in the postcondition of that same compilation goal).
6. The `continuation` premise is another compilation goal, this time for the code that follows the loop; the binding for `v` ensures that the rest of the derivation will treat `v` opaquely (instead of inlining `Nat.iter f a` into the rest of the code).

These design choices are the result of a long series of iterations on loops. Fundamentally, a loop lemma in Rupicola connects an iteration function (which repeats a computation a certain number of times) to a `while` loop in Bedrock2. This loops has an stop condition, typically `index < bound` or `index > 0`, as well as an invariant (which is used to constrain the derivation of the loop body), typically some formula giving exactly the state of the memory and of the locals after a certain number of iterations.[18]

For the equivalence between the Gallina and Bedrock2 versions to hold, we need to know that the loop body preserves the invariant and that the loop runs the right number of times. In particular, we need the loop body to update the loop counter (in the following, assume that the update is an increment). There are a number of ways to design this, with the difficulty being deciding which part of the code is responsible for that update, and hence, and hence how the update is reflected in the invariant:

1. Include the counter increment in the postcondition that the body needs to satisfy, and derive the corresponding Bedrock2 code as part of the derivation of the loop body. This isn't convenient, because the Gallina code does not increment a counter, so there is no Gallina code to drive the derivation of the increment in Bedrock2.
2. Change the Gallina code so that each iteration of the loop returns the next value of the counter. This solves (1.), but it is hard to do it in a way that guarantees termination.
3. Put all loops in the nondeterminism monad, so that loops are really an arbitrary `repeat()` of a given Gallina function, until a given predicate holds. This solves (2.), since the Gallina code does not need to check for termination anymore, but since Bedrock2 requires termination these loops could not be straightforwardly compiled.
4. Force the inclusion of an increment into the Bedrock2 loop body. That is, initialize the evar corresponding to the Bedrock2 version of the Gallina loop body to include an increment: `{{ t; l; m; σ }} ?body; index++ {{ invariant (S i) f }}`. Unfortunately the compiler is not prepared to deal with already-compiled code (the `index++` part following the `?body` evar), and besides there is still no Gallina code matching that increment.
5. Append the increment to the synthesized bedrock2 loop body; use the invariant spe-

[18] In reality the invariant is keyed by the (Gallina-level) iteration counter `index`, so it is not strictly an "invariant". An alternative style would read the value of the index from the local map of variables, but passing in the expected value of the index makes invariants easier to phrase and is in keeping with our desire to keep Gallina-level and Bedrock2-level reasoning separate.

cialized to a map containing an updated index as the postcondition of the synthesized body.

Remember that the issue is to know which post-condition to use to compile the loop body. Here, the idea is to say that the loop body's postcondition is exactly the loop invariant, but applied to a map of locals in which the loop index is incremented. In other words, the code will still be `?body; index++`, but we only reason about `?body` (so we won't run into the issues of (4.)): we give the `?body` a postcondition `{{ invariant (S i) (map.put locals "i" (S i) }}`. Morally, what this is doing is computing the WP of the increment and specializing the loop predicate to that. If we know that this holds after executing the synthesized body, then in particular `invariant` will still hold after executing the increment.

The problem here is the final step of unification, after we finish compiling the loop body. The compiled code will have performed all sorts of locals manipulations, and having to assert `invariant` with an additional layer of `map.put` throws a wrench into the automation. For example, if the body sets then unsets some local variables, then we will find ourselves having to prove `invariant (map.put (map.remove ?keys l) …)`, with the additional difficulty that instantiating `?keys` is in fact part of the final unification process.

6. Do the same as (5.), but do not specialize the postcondition; instead, add constraints on the invariant. Specifically, ensure that the only thing the `invariant` does with the counter is to store it in the right place in the map of locals. This guarantees that `invariant i l` implies `invariant i' (map.put locals "i" i')`; it is what we called *predicate stability* in the discussion of `Nat.iter` above (note how it allows us to have an uncluttered postcondition in the *loop body* premise above).[19]

In Rupicola I used solution (1.) to phrase and prove the most basic loop lemma, and then derive specialized variants of it in the shape of solution (6.).

4.6.2.4 Extensional effects (monads)

Rupicola's standard library implements a writer monad, an I/O monad, a non-determinism monad, and a generic free monad. For each of these we define a monad instance, from

[19]This may seem restrictive, but in fact it has to be true even for solution (5.) to work, since otherwise the final increment would not set the index to the right value.

which we derive a `bindn` constructor that captures binder names as strings in addition to the usual value and continuation.

We then define two lifts: one to state postconditions in function specifications, and one to state postconditions in compilation goals. For the nondeterminism monad, the first one is `ndspec`, and the second `ndspec_k`:

```
Definition ndspec {A} (c: ND.M A) (P: A → Prop) :=
  ∃ a, c a ∧ P a.
Definition ndspec_k {A} (P: A → predicate) (c: ND.M A) : predicate :=
  fun tr mem locals ⇒ ndspec c (fun a ⇒ P a tr mem locals).
```

The lift criterion described in section 4.5.1.2 is a consequence of the same property of `ndspec`:

```
Lemma ndbind_bindn {A B} pred vars (nd: ND.M A) a (k: A → ND.M B):
  nd a →
  ndspec (k a) pred →
  ndspec (mbindn vars nd k) pred.
Lemma WeakestPrecondition_ndspec_k_bindn {A B} c t m l σ post
    vars (nd: ND.M A) a (k: A → ND.M B) :
  nd a →
  {{ t; m; l; σ }} c {{ ndspec_k post (k a) }} →
  {{ t; m; l; σ }} c {{ ndspec_k post (mbindn vars nd k) }}.
```

With these in place we then define a setup lemma that recognizes postconditions that employ the first lift and change them into the second lift, and finally we define lemmas that provide support for compiling monadic operations, such as stack allocation above, or in the case of the nondeterminism monad calls to any nondeterministic Bedrock2 functions.

4.6.2.5 Low-level features

Rupicola has intensional encodings of a variety of low-level Bedrock2 features. I discuss inline tables and stack allocation in detail in two case studies in section 5.1.2. Rupicola also supports mapping Gallina functions to arbitrary Bedrock2 functions, to be separately compiled and linked.

4.6.3 Unsupported features and limitations

Rupicola does not aim to support all of Gallina, so incompleteness is one of its fundamental limitations: its input language depends on the set of compilation lemmas provided by the user.

This is not a significant issue in most cases, but it does mean that a lot of the facilities that come for free in Gallina must be translated into individual compilation lemmas. A prime example of this is matches: in Gallina pattern matching is automatically available on all user defined types, whereas in Rupicola pattern matching is not available on any types out of the box (and, as it is a control-flow construct, it requires special care).

Other limitations stem from limitations in Coq's pattern-matching, as well as limitations in Bedrock2. For example, Bedrock2 does not support arbitrary recursion (to rule out stack overflows), and neither does Rupicola (an additional wart in implementing support for raw Gallina `fix` constructs is that, like `let`-bindings, a Coq lemma cannot capture the body of a `fix`; but unlike `let` bindings where we can define the `nlet` combinator, for `fix` we would need a general fixpoint combinator, which is not expressible in Coq). Similarly, Bedrock2 distinguishes statements and expressions, and that design leaks into Rupicola (we need to guess whether a particular binding needs a statement compilation lemma or can be compiled as an expression).

Beyond this, the main pain point with Rupicola is compilation speed, which I discuss in section 6.1.

4.7 Rupicola step-by-step

To tie everything together, let us revisit our original example one last time, this time peeking under the hood of the `compile` tactic.

Recall that in section 4.1 we defined specifications `upstr` and `toupper`, lowered functional implementations `upstr'` and `toupper'`, as well as a signature; here I reproduced only `upstr'` and `toupper'`:

```
Definition upstr' (s: list byte) :=
  let/n s := ListArray.map
               (fun b ⇒ byte_of_ascii (toupper (ascii_of_byte b)))
               s in s.

Definition toupper' (b: byte) :=
  if byte.wrap (b - "a") <? 26
  then byte.and b x5f else b.
```

Let us now see what happens when we run `compile`. That tactic is defined to perform three steps: `compile_setup`; `repeat compile_step`; and finally `compile_done` (the last step only checks whether there are any compilation goals left, and if so shows some hints to the user). For brevity in all goals below `|s|` stands for `(length s)` and casts from `nat`, `byte`, and `Z` to word are implicit in values in local variables maps (so for example `#{ ... m; k ⇒ |s| }#` is short for `map.put m k (of_nat (length s))`).

```
Derive upstr_br2fn SuchThat
  (defn! "upstr"("s", "len") { upstr_br2fn },
   implements upstr') As upstr_br2fn_ok.
Proof.
```

The compile_setup tactic unfolds the `defn!` notation and refactors the post-condition of the compilation relation to make it explicit which program we are compiling (based on the `implements` clause in the call to `Derive`. For readability we omit uninteresting hypotheses, but notice how hypothesis `H0` captures all the information on the variables mentioned in the precondition of the program:

```
compile_setup.
```

At this point `compile_step` performs some cleanups. To next lemma to apply is `compile_byte_listarray_map` lemma, but to be able to call it we need to (1) change the goal to an nlet_eq goal; (2) infer a loop predicate; and (3) pick a name for temporary variables. `compile_step` does (1); the `compile_map` tactic does (2) using the invariant inference infrastructure and (3) using a simple gensym. We can peek at the results by invoking the loop-inference logic manually:

```
(fun (idx : Z) (args : list byte) (tr' : Semantics.trace)
   (mem' : BasicC64Semantics.mem) (locals' : locals) ⇒
 tr' = tr ∧
 locals' = #{ "s" ⇒ p; "len" ⇒ wlen; "idx" ⇒ idx; "bound" ⇒ |s| }# ∧
 (nbytes |s| p args ⋆ r) mem')
```

Apply the loop compilation lemma produces a number of goals, most of which are trivial side conditions; for concision I omit most of them here:

```
apply compile_nlet_as_nlet_eq; compile_map. (* ... *)
```

```
H3: Z.of_nat |s| < 2 ^ 64
______________________________________
0 ≤ Z.of_nat |s| < 2 ^ 64
```

```
H1: wlen = of_nat |s|
______________________________________
DEXPR mem #{ "s" => p; "len" => wlen; "_from" => 0 }# ?Goal (of_nat |s|)
```

```
______________________________________
{{ tr; mem0; #{ "s" => p; "len" => wlen; "_from" => idx; "_to" => |s| }#; σ }}
  ?Goal0
{{ lp idx
     (let/n tmp as "_tmp" := ListArray.get a idx in
      let/n tmp0 as "_tmp" := byte_of_ascii (toupper (ascii_of_byte tmp)) in
      let/n x as "s" := ListArray.put a idx tmp0 in x) }}
```

```
______________________________________
{{ tr; mem0; #{ "s" => p; "len" => wlen; "_from" => |s|; "_to" => |s| }#; σ }}
  ?Goal1
{{ pred v }}
```

The first goal captures the fact that the size of the array is representable as a 64-bits integer (this compilation run assumes a 64-bits machine), which we assumed as part of the function's precondition; the second one needs us to provide an expression that reduces to the length of s; the third, a program that implements the body of the loop; the fourth, a program to implement the continuation of the original program, which does nothing.

```
4: solve [repeat compile_step].
1: lia.
```

The expression compilation goal is straightforward, since the variable "len" contains the value we need.

```
subst wlen; reify_change_dexpr_locals.
```

```
______________________________________
DEXPR mem (map.of_list [("_from", of_Z 0); ("len", of_nat |s|); ("s", p)])
  ?Goal (of_nat |s|)
```

```
apply expr_compile_var.
```

```
______________________________________
map.get (map.of_list [("_from", of_Z 0); ("len", of_nat |s|); ("s", p)]) ?s =
Some (of_nat |s|)
```

```
expr_instantiate_map_get.
```

```
________________________________________________________________
map.get (map.of_list [("_from", of_Z 0); ("len", of_nat |s|); ("s", p)]) "len" =
Some (of_nat |s|)
```

```
apply map.get_of_str_list_assoc_impl.
```

```
________________________________________________________________
map.list_assoc_str "len" [("_from", of_Z 0); ("len", of_nat |s|); ("s", p)] =
Some (of_nat |s|)
```

```
reflexivity.
```

Then we move on to a more interesting part, compiling the body of the loop. The first step is to compile the array lookup; its side conditions are that we (1) have access a pointer to an array whose contents match that of the list we are reading from, as evidenced by a separation logic predicate; (2) have a way to compute an expression yielding that pointer; (3) have a way to compute the index being accessed; and (4) that we index be in bounds. After that, we are left with a single continuation goal, corresponding to the rest of the program:

```
apply compile_nlet_as_nlet_eq; eapply compile_byte_sizedlistarray_get.
```

```
________________________________________________________________
(nbytes ?len ?a_ptr a ⋆ ?R) mem0
```

```
________________________________________________________________
DEXPR mem0 #{ "s" => p; "len" => |s|; "_from" => idx; "_to" => |s| }#
  ?Goal ?a_ptr
```

```
________________________________________________________________
DEXPR mem0 #{ "s" => p; "len" => |s|; "_from" => idx; "_to" => |s| }#
  ?Goal0 (of_Z idx)
```

```
________________________________________________________________
idx < Z.of_nat ?len
```

```
________________________________________________________________
{{ tr; mem0;
   #{ "s" => p; "len" => |s|; "_from" => idx; "_to" => |s|; "_tmp" => v }#; σ }}
  ?k_impl
{{ lp idx
     (let/n tmp as "_tmp" := byte_of_ascii (toupper (ascii_of_byte v)) in
      let/n x as "s" := ListArray.put a idx tmp in x) }}
```

```
1-4: solve [repeat compile_step].
```

At this point we can rewrite the unoptimized toupper into its optimized counterpart and inline the result, which we did using `Hint Rewrite` and `Hint Unfold` in the automated version:

```
rewrite toupper'_ok; unfold toupper'.
```

```
________________________________________________________________________
{{ tr; mem0;
   #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v }#; σ }}
  ?k_impl
{{ lp idx
     (let/n tmp as "_tmp" :=
        if byte.wrap (v - "a"%byte) <? 26 then byte.and v "_" else v in
      let/n x as "s" := ListArray.put a idx tmp in x) }}
```

The result is a conditional, which requires a similar predicate inference trick as we saw previously; for concision I omit the details and use the `compile_if` tactic directly. This time we find ourselves with four goals: one for the test expression; two for the branches of the if; and one for its continuation:

```
apply compile_nlet_as_nlet_eq; compile_if.
```

```
________________________________________________________________________
DEXPR mem0
  #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v }#
  ?c_expr (word.b2w (byte.wrap (v - "a"%byte) <? 26))
```

```
________________________________________________________________________
{{ tr; mem0;
   #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v }#; σ }}
  ?t_impl
{{ val_pred (let/n x as "_tmp" := byte.and v "_" in id x) }}
```

```
________________________________________________________________________
{{ tr; mem0;
   #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v }#; σ }}
  ?f_impl
{{ val_pred (let/n x as "_tmp" := v in id x) }}
```

```
________________________________________________________________________
{{ tr0; mem1; locals; σ }}
  ?k_impl
{{ lp idx (let/n x as "s" := ListArray.put a idx v0 in x) }}
```

The first three goals are solved by the expression compiler; for succinctness I show details only for the first one. The purpose of the conversion to `map.of_list` that the first tactic performs is to speed up later calls to expr_compile_var.

```
{ apply expr_compile_Z_ltb_u.
```

```
________________________________________
DEXPR mem0
  #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v }#
  ?e1 (of_Z (byte.wrap (v - "a"%byte)))
________________________________________
DEXPR mem0
  #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v }#
  ?e2 (of_Z 26)
________________________________________
0 ≤ byte.wrap (v - "a"%byte) < 2 ^ 64
________________________________________
0 ≤ 26 < 2 ^ 64
```

```
  - apply expr_compile_byte_wrap.
```

```
________________________________________
DEXPR mem0
  #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v }#
  ?e1 (of_Z (Z.land (v - "a"%byte) 255))
```

```
    apply expr_compile_Z_land.
```

```
________________________________________
DEXPR mem0
  #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v }#
  ?e1 (of_Z (v - "a"%byte))
________________________________________
DEXPR mem0
  #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v }#
  ?e20 (of_Z 255)
```

```
    + apply expr_compile_Z_sub.
```

```
________________________________________
DEXPR mem0
  #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v }#
  ?e1 (of_byte v)
________________________________________
DEXPR mem0
  #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v }#
  ?e21 (of_byte "a"%byte)
```

```
        * reify_change_dexpr_locals.
```

```
  ____________________________________________________________
  DEXPR mem0
    (map.of_list
       [("_tmp", of_byte v); ("_to", of_nat |s|); ("_from", of_Z idx);
       ("len", of_nat |s|); ("s", p)]) ?e1 (of_byte v)
```

```
          expr_compile_var.
        * apply expr_compile_Z_literal.
      + apply expr_compile_Z_literal.
    - apply expr_compile_Z_literal.
    - eapply byte_range_64.
    - lia. }

  1,2: solve [repeat compile_step].
```

As for the last goal, all that remains is to compile `ListArray.put` into a write into a pointer; the reasoning is very similar to the `ListArray.get` case, so I omit it:

```
  ____________________________________________________________
  {{ tr0; mem1; locals; σ }}
    ?k_impl
  {{ lp idx (let/n x as "s" := ListArray.put a idx v0 in x) }}
```

```
  apply compile_nlet_as_nlet_eq;
    eapply compile_byte_sizedlistarray_put. (* ... *)
```

```
  ____________________________________________________________
  {{ tr; mem';
     #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s|; "_tmp" ⇒ v0 }#; σ }}
    ?k_impl
  {{ lp idx v1 }}
```

This final goal simply requires unifying the postcondition and the precondition (i.e. hypotheses in the context):

```
  apply compile_unsets with (vars := ["_tmp"]).
```

```
  ____________________________________________________________
  {{ tr; mem'; #{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s| }#; σ }}
    ?Goal
  {{ lp idx v1 }}
```

```
  apply compile_skip.
```

```
______________________________________________________________________
tr = tr ∧
#{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s| }# =
#{ "s" ⇒ p; "len" ⇒ |s|; "_from" ⇒ idx; "_to" ⇒ |s| }# ∧
(nbytes |s| p v1 ⋆ r) mem'
```

```
  tauto.
Qed.
```

5 Evaluation

I claim that Rupicola's novelty is its combination of extensibility, foundational proofs, and performance. The first and third claims are measurable. To support them, I evaluated Rupicola from three angles: programmer experience, expressivity, and performance. For the first two I used case studies, and for the third, performance benchmarks.

5.1 Programmer experience and expressivity

5.1.1 Case study: Extending Rupicola

Extending a traditional compiler can be a daunting task: compilers sometimes supports extensions of a very restricted kind (e.g. single-language rewrites), but these are not sufficient: for Rupicola to generate code whose performance matches that of handwritten programs we need users to be able to plug in new translation strategies, new logic, and new decision procedures.

Implementing such complex extensions in a traditional compiler would be a daunting task: it would typically require writing new compilation passes or extending existing ones by directly modifying the implementation of the compiler itself. Rupicola is intended to make this much easier, and this section gives some evidence to that effect.

Anecdotal observation suggests that the corresponding effort in Rupicola is minimal: once users develop sufficient familiarity with our framework, they find it manageable to teach the compiler new lemmas to support the patterns that they are interested in (I have observed this anecdotally as more students became involved in this project). I summarize some examples of estimated effort in development time and lines of code in table 5.1.

Adding support for new monads is also straightforward, though naturally a bit more complicated. As a concrete example, I estimate that adding support for a writer monad starting from a blank file required about an hour and a half, with a bit over 15 minutes

Domain	Operation	Lemma	Proof	Time (min)
`nondet`	`alloc, peek`	26+24	17+11	13+6
`cells`	`get, put`	22+23	5+3	7+3
	`iadd`	31	7	8
`io`	`read, write`	25+26	7+10	11+8

Table 5.1: Verification effort for user extensions. Time estimates are rough indicators. "Lemma" and "Proof" refer to line counts using a verbose notation; a more succinct notation would shrink the "Lemma" column by about 30%.

spent defining the monad and proving its properties (17 lines of code, 5 lines of proofs), 30 minutes spent setting up the compilation of that monad (56 lines of code, 8 lines of proofs), 20 minutes to add a Gallina primitive and compilation lemmas for it (mapping writes to I/O trace operations at the Bedrock2 level; 50 lines of code, 15 lines of proofs), 15 minutes to write a small example and compile it (4 lines of Gallina model, 6 lines for the Bedrock2 signature, and 1 line for the compilation "proof": `compile.`), and about 3 seconds to derive the actual code[20]. The same example written by imitating other monad examples would probably take roughly a third to half as long.

5.1.2 Case study: Implementing compiler extensions to support new low-level patterns

Section 4.5 above describes in detail two of the most significant compiler extensions that we implemented. Here I give evidence of Rupicola's usability through two additional examples, both of which are instances of one of the most interesting kinds of extensions for a relational compiler: extensions that expose in the shallow world features of the low-level language. Specifically, I look at stack allocation and inline tables.

Implementation efforts for both of these extensions were led by Dustin Jamner, with extensions by the author of this thesis; I report on them here as further evidence that Rupicola is usable by experienced Coq users, and as interesting case studies of Rupicola extensions.

[20]While the programs that Rupicola produces are fast, Rupicola itself is not. I give more details about this in section 6.1.

5.1.2.1 Stack allocation

Bedrock2 supports (lexically scoped) stack allocations: a block of code can be wrapped in a binding construct giving it access to a pointer to a block of compile-time constant-size memory allocated on the stack. This is particularly useful for any program that needs access to a small working area, and unlike a global buffer it does not pollute external specifications (beyond changing the function's stack-space requirements, which Bedrock2 tracks).

As a case study (and because a larger development using Rupicola was planning to make use of it), we extended Rupicola to provide access to this feature in Coq programs. We exposed it under three APIs.

- For allocating space intended to be fully initialized by a single operation or function call, we use a semantically transparent annotation, `stack`. When Rupicola sees `let/n x := stack (term) in ...`, it generates a stack allocation in Bedrock2 and resumes compilation with the plain program `let x/n := term in ...` and a memory context containing an uninitialized block of memory pointed to by `"x"`. The determination of how large a block to allocate is done by looking up an instance of a type class. Then, the compilation of `term` is expected to initialize the allocated block fully.
- For allocating space that is expected to be fully initialized in multiple small steps, we use the buffer API discussed in section 4.6.2.2, with consecutive calls to push followed by a final call to convert the buffer into an array. The idea generalizes: we construct a separation logic predicate that captures the initialized parts of the allocated space, with APIs that progressively grow the allocated section, and finally an API to convert from that predicate to a simpler one that applies only to fully allocated value.
- For allocating uninitialized space that is not expected to be fully initialized right away, we use a monadic computation returning a list of fixed length containing arbitrary values in Gallina, and we map this to a stack allocation of the same width in Bedrock2. In cases where possible to prove that a monadic computation involving stack allocations is in fact deterministic (e.g. because the code in fact overrides all positions in the nondeterministic list that the allocation returns), then the nondeterminism could be restricted to the code fragment performing the allocation, and as a result the local effect does not leak.

5.1.2.2 Inline tables

Inline tables are another Bedrock2 feature that is usefully exposed at the functional level; they are const arrays local to a Bedrock2 function, useful for implementing lookup and translation tables.

The Gallina API that we implemented is the same as that of arrays, except that only one operation (`get`) is available. Crucially, the API does not impede reasoning about the code: simply unfolding the definition of `InlineTable.get` reveals that it is just the function `nth` on lists.

The Gallina API accepts any type in the array; it is the user's responsibility to then show how these values can be cast to a scalar type (bytes or machine words). This flexibility makes it possible to store values of types such as `Fin.t` (a subset integer type equivalent to `{i | i < n }`) in an inline table (as long as `n` is less than 256 for a byte or 2^{width} for a word): we use this in the UTF8 decoder to store arrays of indices into another array, which makes it trivial to statically encode the fact that all values in the first array are valid indices into the second array.

We have Rupicola compilation lemmas to load either a single byte or a full machine word from these tables. Due to the way the semantics of Bedrock2 are written, the effort for machine words is much greater than the effort for the bytes version of the same lemmas (hundreds versus tens of lines). Most required lemmas are dues to an idiosyncrasy of the semantics of inline tables: they are about basic properties of an otherwise seldom-used set of Bedrock2 functions and are irrelevant to Rupicola (the plan is to offer these lemmas to the authors of Bedrock2 for merging into that repository, which will bring the longer proof back to tens of lines).

5.1.3 Case study: Rupicola's expression compiler

This section and the next attempt to give a sense of the effort involved in developing and using relational compilers.

Rupicola is really two relational compilers rolled into one: one targeting Bedrock2's statements and one targeting its expressions. Originally, however, we assumed that the expression part of the compilation process was so simple that it would not warrant the cost of relational compilation. Instead, we compiled expressions by reifying them into an AST type and then using a very simple verified compiler targeting Bedrock2's expression language, and we planned to handle all necessary extensions by plugging in new cases in our reflection tactics and proofs.

We were wrong; over time this reflective compiler grew more complicated and accumulated various complications to make it easier to extend to more types and more operations; and we needed constant extensions to it, because programmers use a wide range of numeric types in Gallina and expect to be able to map operations on them to low-level expressions (rather than to a sequence of individual statements, each performing a single operation).

I switched to a relational compiler at a time when the reflective compiler had support only for machine words, operations on Z, and a limited subset of Boolean operations. The original reflective compiler was written with conciseness in mind, requiring about 450 lines of code (including 200 lines of tactics and 60 lines of typeclass definitions) and would (we estimated) require about 100 more lines of tactics, 100 lines of proofs, and about the same amount in refactoring to existing tactics to add support for byte operations. In addition, the implementation was fairly technical, so extensions had to be handled by someone intimately familiar with the code base.

The relational compiler that I replaced it with was about 250 lines of code (of which about 30 `Hint` commands to assemble the lemmas into a compiler) and then quickly grew to about 400 lines to support bytes, Booleans, integers, two representations of natural numbers, and mixed expressions (with casts between different types). None of these extensions required deep expertise; in fact, shown below is *all* of the code that we needed to support `byte.and`, the change that daunted us into switching to relational compilation:

```
Lemma expr_compile_byte_and
      (m: mem) (l: locals) (b1 b2: byte) (e1 e2: expr) :
  DEXPR m l e1 (of_byte b1) →
  DEXPR m l e2 (of_byte b2) →
  DEXPR m l (expr.op and e1 e2) (of_byte (byte.and b1 b2)).
Proof. rewrite byte_morph_and; apply expr_compile_word_and. Qed.

Hint Extern 1 ⇒ simple eapply expr_compile_byte_and; shelve : compiler.
```

Even better, the switch to relational compilation allowed plugging in support for transformations with complex side conditions trivially — operations such as arithmetic shifts[21] or array dereferences[22], for example.

[21]Our original reification-based expression compiler supported left shifts but not right ones, since they have preconditions.

[22]Even with more elaborate infrastructure to generate side conditions, we have sufficiently many represen-

Surprisingly, the performance cost (on compilation times, not on performance of compiled programs) was never more than two times, yielding an overall 30% slowdown in the worst case; significant, but smaller than I feared given the performance benefits I had hoped for when originally adopting a reflective approach.

5.1.4 Case study: End-to-end verification with Rupicola

Narrowly speaking, the exact techniques that programmers employ to generate input suitable for compilation with Rupicola is out of the scope of this thesis: Rupicola is built to be agnostic to this. Authors that find themselves missing a feature may chose to implement compiler extension to support it (confident in the knowledge that they will not break the rest of the compiler) or may choose to lower these unsupported constructs into ones that Rupicola does support, if their development process lends itself to that (in a refinement-based pipeline, for example). As a concrete example, when this paragraph was originally written, Rupicola had support for left folds on lists but not right ones: a programmer employing the later could have proven a new compilation lemma, or if their program permitted it, lowered it either to a left-fold variant or to one of the more basic iteration primitives that Rupicola supported (most likely iterating over a range of numbers, retrieving the n-th element of the list after each iteration).

Still, asking authors to lower their programs leaves the question of the expressivity of Rupicola's input language: how easy is it to massage high-level functional programs into ones that Rupicola will accept? In general, I have found it very easy, for two reasons. First, all reasoning happens between shallowly embedded programs, so the verification experience is one that interactive theorem provers excel at: proving equivalences between relatively small pure functions that operate on inductive data types. Second, in many cases, the lowering that Rupicola requires is really a form of transparent or semitransparent program annotation: for example, we specify which object we intend to mutate by using a variant of `let` annotated with a variable name, or we specify that a particular object should be stack-allocated by wrapping its initial value with the `stack` function. Because these annotations are semantically irrelevant (they are just identity function with extra arguments), unfolding them returns to the original program.

tation of arrays that it would have been daunting to encode each of them in the reflective compiler; as a result early versions of Rupicola required let-binding array accesses separately, to invoke a separate compilation lemma.

To illustrate these points on a concrete example, I implemented and verified the TCP/IP checksum algorithm, a one's-complement sum of 16-bit values (unsigned addition with carries added back in).

I chose this specific program because it proved particularly vexing in previous work: when we implemented this function in the context of Narcissus [12], we did not manage, even with a careful (and unverified) extraction setup, to achieve satisfactory performance for it. Instead, we resorted to replacing it as a whole by an unverified but sufficiently fast implementation in OCaml.

For this instance, I sought to make the specification as readable and easily auditable as possible (for example, instead of relying on a bitvector type, I used Coq's built-in byte and Z types).

```
Definition onec_add16 (z1 z2: Z) :=
  let sum := z1 + z2 in
  (Z.land sum 0xffff + (Z.shiftr sum 16)).

Definition ip_checksum (bs: list byte) :=
  let c := List.fold_left onec_add16
    (List.map le_combine (chunk 2 bs))
    0xffff in
  Z.land (Z.lnot c) 0xffff.
```

There is a subtlety in the program above: even though IP checksums are defined on 2-byte blocks, the input may contain an odd number of octets. The specification handles this case gracefully (the last chunk that it receives is only one byte long).

I then separately wrote a Rupicola-ready version of the same program (next page); this version separates iteration over the even-length prefix of the input and the optional last byte. In a more traditional development workflow, this is the functional model that one would verify a handwritten low-level implementation against (direct verification against the high-level spec would be possible but would involve all the complexity of the two-step process forced into a single larger proof).

Proving the equivalence of these two versions is straightforward and proceeds in two steps, none of which use lemmas particularly specific to IP checksums. First, we show that we can remerge the loop and the last conditional step; this is because the nth function that we use in the loop returns a default value if we ask for an out-of-bounds value[23]. Then, we translate the remerged loop on an integer range into a fold, and from there, the proof is straightforward. The total amount of proofs that is specific to IP checksums is a few tens of lines, including some properties of the original programs to guarantee that one's-complement sums fit in the finite types that we use in the implementation[24].

```
Definition ip_checksum_upd (c16: word) (b0 b1: byte) :=
  let/n w16 := b0 |w (b1 <<w 8) in
  let/n c17 := c16 +w w16 in
  let/n c16 := (c17 &w 0xffff) +w (c17 >>w 16) in
  c16.

Definition ip_checksum' (bs: list byte) : word :=
  let/n c16 := 0xffff in
  (* Main loop *)
  let/n c16 := nd_ranged_for_all
    0 (Z.of_nat (length bs) / 2)
    (fun c16 idx =>
       let/n b0 := ListArray.get bs (2 * idx) in
       let/n b1 := ListArray.get bs (2 * idx + 1) in
       let/n c16 := ip_checksum_upd c16 b0 b1 in
       c16) c16 in
  (* Final iteration *)
  let/n c16 := if Nat.odd (length bs)
     then let/n b0 := ListArray.get bs (length bs - 1) in
          ip_checksum_upd c16 b0 x00
     else c16 in
  (* Clean up *)
  let/n c16 := (~w c16) &w 0xffff in
  c16.
```

Finally, I wrote a signature for the function and ran the compiler. It takes a few seconds for the derivation to complete, and it requires a single extension: a lemma to prove that if $n \geq 0$ is odd, then $n - 1 \geq 0$, which is required to prove that the final array access (at `length bs - 1`) is in bounds.

[23]Why not use the merged version as the input to Rupicola, then? Because we do not want to perform explicit bounds checks in the generated code. In the separated version, we can trivially prove that all accesses are in-bounds as part of the compilation process, whereas the merged version potentially performs an out-of-bounds access, which we could only compile with a conditional that would slow all iterations of the loop, not just the last.

[24]Take these numbers with a grain of salt, as they are easy to manipulate and hard to interpret. For example, I had to prove that `Z.odd (Z.of_nat n) = Nat.odd n` and that an odd number equals one modulo two: I did not consider these facts to be "specific to IP checksums", but I did have to spend time proving them, since they were missing from Coq's standard library. Time spent is a more reliable metric.

```
Instance spec_of_ip_checksum : spec_of "ip_checksum" :=
  fnspec! "ip_checksum" data_ptr wlen / data R ~> chk, {
    requires tr mem :=
      wlen = of_nat (length data) /\
      (bytes data_ptr data ⋆ R) mem;
    ensures tr' mem' :=
      tr' = tr /\ chk = ip_checksum' data /\
      (bytes data_ptr data ⋆ R) mem'
  }.
```

Overall, implementing and verifying IP checksums in Rupicola was a matter of a few hours, and the resulting performance is on par with a hand-coded version in C — but with proofs!

5.1.5 Case study: Third party contributions

Beyond the examples in its own repository, Rupicola is currently being used by members of the Fiat-Crypto project [13, 43, 15] to extend the framework beyond individual finite field operations. This section briefly summarizes these efforts to give a sense of developments in Rupicola that have happened beyond my own efforts.

5.1.5.1 Montgomery ladder

Starting from code written by Jade Philipoom, Dustin Jamner is compiling an implementation of the Montgomery ladder, an algorithm for constant-time multiplication on elliptic curves [33, 4]. The implementation calls out to separately compiled Fiat-Crypto functions for arithmetic operations. It requires a number of Rupicola extensions, including for compiling individual arithmetic operations (that code is parametric on the choice of field parameters) and support for stack allocation.

Below is the input that is fed to Rupicola. Its main loop calls a separate Gallina function `ladderstep` — it, too, is compiled using Rupicola.

```
Definition montladder (sz: nat) (testbit: nat → bool) (u: E) : E :=
  let/n X1 := stack 1 in let/n Z1 := stack 0 in
  let/n X2 := stack u in let/n Z2 := stack 1 in
  let/n swap := false in
  let/n (X1, Z1, X2, Z2, swap) :=
     iter_down sz (fun i '⟨X1, Z1, X2, Z2, swap⟩ ⇒
         let/n s_i := testbit i in
```

```
          let/n swap := xorb swap s_i in
          let/n (X1, X2) := cswap swap X1 X2 in
          let/n (Z1, Z2) := cswap swap Z1 Z2 in
          let/n (X1, Z1, X2, Z2) := ladderstep u X1 Z1 X2 Z2 in
          let/n swap := s_i in
          ⟨X1, Z1, X2, Z2, swap⟩)
        ⟨X1, Z1, X2, Z2, swap⟩ in
    let/n (X1, X2) := cswap swap X1 X2 in
    let/n (Z1, Z2) := cswap swap Z1 Z2 in
    let/n r := Z1 ^ -1 in
    let/n r := X1 * r in
    r.
```

One feature that shines in this use case is Rupicola's handling of mutation and allocation: while the code reads like regular Gallina code, the names chosen for the let bindings and the calls to stack indicate which variables should be mutated and when allocation should be performed. When reasoning about the code, simply unfolding the definitions of nlet and stack erases all traces of Rupicola, and as a result the proof connecting this lowered implementation to the pure Gallina spec is a matter of just a few lines.

This ladderstep function plugs into a larger derivation, which also serves as a useful case study of the convenient of our approach to loops. The original implementation of used a complex loop lemma that required the user to define ghost state and a custom invariant, and consequently the compilation process was only partly automated: in fact, the first implementation of the derivation was close to 200 lines of a complex mix of domain-specific automation and manual proofs, plus roughly 50 lines of invariant definitions and 80 line of additional tactics. We performed three simplifications:

1. Use a better (more structural) encoding of the relation between modular bignums and integers: in memory bignums are arrays of bytes, which correspond to a certain integer value modulo some constant M. The original derivation used a separation logic predicate that made the byte representation of bignums explicit, and as a result all arithmetic has to reason about casts between byte arrays and integers. We introduced a new separation logic predicate to encapsulate the correspondence, essentially hiding the interpretation of bytes under an existential (`bignum z := ∃ bs: list bytes, eval bs = z mod M`), as well as compilation lemmas to translate modular arithmetic on Z to operations on these new bignums (the actual predicate also hides encoding details related to bounds on these bignums).

2. Use automatic predicate inference: the previous change was enough to eliminate all need for ghost state and custom invariants. Had it not been, we would have been able to prove properties about partial runs of the loop body *at the Gallina level* and use these as part of the derivation, still without having to reason at the Bedrock2 level.
3. Use `nlet` bindings to drive mutation vs. allocation decisions (the original implementation used annotations on the separation logic predicates). This did not require much effort: the original core already mostly conformed to Rupicola's assumptions about naming, even though it did not use `nlet`.

The result is a one-line derivation (`compile.`) supported by a handful of new bignum compilation lemmas, about 10 lines of logic to drive the application of one of these lemmas, and a few additional compilation hints to plug in rewrites and a linear arithmetic solver. As a bonus, the new version of the code does more than the original: unlike its predecessor, it takes care of allocating its own scratch space instead of requiring it to be passed in.

5.1.5.2 Modular exponentiation

Separately, I have been helping Ashley Lin derive optimized code for in-place modular exponentiation with known exponents using Rupicola, using a multiply-and-square algorithm. Her implementation is verified from end to end.

The derivation starts from a high-level specification of exponentiation. The code uses an axiomatic specification of field elements that states `to_Z (pow x n) = (to_Z x) ^ n mod M` (where `to_Z` converts a field element to `Z` and `pow` is the field exponentiation), so here the entire specification is simply `pow x n`.

The lowering phase is semi-automatic. We start with a traditional multiply-and-square algorithm defined by induction over the binary decomposition of the exponent (below, the type `positive` is a binary encoding of positive natural numbers, `…~0` matches numbers whose lowest bit is 0, and `…~1` matches numbers whole lowest bit is `1`). Its proof of correctness is trivial by induction.

```
Fixpoint pow_sq (x: E) (n: positive) : E :=
  match n with
  | 1   ⇒ x
  | n~0 ⇒ let/n r := pow_sq x n in
          r ^ 2
  | n~1 ⇒ let/n r := pow_sq x n in
```

```
         let/n r := r ^ 2 in
         r * x
  end.

Lemma pow_sq_ok (x: E) n:
  pow_sq x n = x ^ (Z.pos n).
Proof.
  induction n; simpl; unfold nlet; rewrite ?IHn.
```

```
────────────────────────────────────────
(x ^ Z.pos n) ^ 2 * x = x ^ Z.pos n~1

────────────────────────────────────────
(x ^ Z.pos n) ^ 2 = x ^ Z.pos n~0

────────────────────────────────────────
x = x ^ 1
```

```
  all: [> rewrite ← pow_sqm_distr |
          rewrite ← pow_sq_distr |
          rewrite pow_1 ].
```

```
x: E     n: positive     IHn: pow_sq x n = x ^ Z.pos n
────────────────────────────────────────
x ^ Z.pos n~1 = x ^ Z.pos n~1

x: E     n: positive     IHn: pow_sq x n = x ^ Z.pos n
────────────────────────────────────────
x ^ Z.pos n~0 = x ^ Z.pos n~0

x: E
────────────────────────────────────────
x = x
```

```
  all: reflexivity.
Qed.
```

Unfolding this definition for a specific exponent yields a tree of nested let bindings; for example, for `n := 71` (and naturally, unfolding the `let/n`s yields the expected Gallina code):

```
Eval simpl in fun x ⇒ pow_sq x 71.
```

```
= fun x : E ⇒
  let/n r as "r" :=
    let/n r as "r" :=
      let/n r as "r" :=
        let/n r as "r" :=
          let/n r as "r" := let/n r as "r" := x in r ^ 2 in r ^ 2 in
        r ^ 2 in
      let/n r0 as "r" := r ^ 2 in r0 * x in
    let/n r0 as "r" := r ^ 2 in r0 * x in
  let/n r0 as "r" := r ^ 2 in r0 * x
: E → E
```

```
Eval cbv in fun x ⇒ pow_sq x 71.
```

```
= fun x : E ⇒ (((((x ^ 2) ^ 2) ^ 2) ^ 2 * x) ^ 2 * x) ^ 2 * x
: E → E
```

A subsequent phase of rewriting then transforms that definition into one that is suitable for plugging in into Rupicola:

```
Eval simpl in (fun x ⇒ lower! pow_sq x 71).
```

```
= fun x : E ⇒
  let/n x0 as "r" := x ^ 2 in
  let/n x1 as "r" := x0 ^ 2 in
  let/n x2 as "r" := x1 ^ 2 in
  let/n x3 as "r" := x2 ^ 2 in
  let/n x4 as "r" := x3 * x in
  let/n x5 as "r" := x4 ^ 2 in
  let/n x6 as "r" := x5 * x in let/n x7 as "r" := x6 ^ 2 in x7 * x
: E → E
```

Minimal extensions to Rupicola are needed to compile these programs: we simply reuse lemmas developed for the Montgomery ladder to map field arithmetic to Bedrock2 primitives.

5.1.5.2.1 Run-length encoded modular exponentiation The naive implementation above generates an amount of code proportional to $\log_2(n)$ for an exponent n. For numbers whose binary representation contains long runs of zeros or ones, a better implementation is possible: we can use a loop to perform the corresponding operation (either square or square-and-multiply) as many times as the corresponding digit (`0` or `1`) is repeated. For example, for `n := 71`, the code above performs three *square* operations followed by three *square and multiply* operations, which we can compress using two loops (`Nat.iter n f x` applies the function f n times to x):

```
Eval simpl in (fun x ⇒ lower! pow_rl x 71).
```

```
= fun x : E ⇒
  let/n x0 as "r" := x ^ 2 in
  let/n x1 as "r" := Nat.iter 2 (fun r : E ⇒ r ^ 2) x0 in
  let/n x2 as "r" := Nat.iter 3 (fun r : E ⇒ r ^ 2 * x) x1 in x2
: E → E
```

Careful readers will notice that we perform a square operation before the first loop. This is to avoid copying the input x into r before the first loop: instead we can square x directly into the output argument r and thereby save a copy.

This code can similarly be fed to Rupicola, this time with an additional lemma to map `Nat.iter` to a loop (section 4.6.2.3). For large exponents, this yields significant space savings (we are using this approach to implement modular inversion, which corresponds in the field that we are working with to exponentiation by $2^{255} - 17$):

```
Eval simpl in (fun x ⇒ lower! pow_rl x (2 ^ 255 - 17)).
```

```
= fun x : E ⇒
  let/n x0 as "r" := x ^ 2 * x in
  let/n x1 as "r" := Nat.iter 248 (fun r : E ⇒ r ^ 2 * x) x0 in
  let/n x2 as "r" := x1 ^ 2 in
  let/n x3 as "r" := Nat.iter 4 (fun r : E ⇒ r ^ 2 * x) x2 in x3
: E → E
```

Proving the correctness of `pow_rl` for all inputs is relatively straightforward, but it is not even necessary. instead, we can just recycle the proof of correctness of `pow_sq`, applied to the specific constant that we are interested in. It suffices because `pow_rl` executes operations in the exact same order as `pow_sq`:

```
Goal forall x, pow_rl x (2 ^ 255 - 17) = x ^ (Z.pos (2 ^ 255 - 17)).
Proof. intros; rewrite ← pow_sq_ok. reflexivity. Qed.
```

5.1.5.3 Future work

These efforts are intended to converge and eventually produce efficient implementations of Poly1305 and ChaCha20 verified from end to end. Andres Erbsen wrote high-level specifications, and we jointly wrote a lowered version in Rupicola, which I proved against the original specifications (some of that code appears in section 4.3). Work is in progress to compile that code to Bedrock2 using Rupicola.

5.2 Performance benchmarks

Flexibility and extensibility are not the only metric that Rupicola is optimizing for: both are intended to enable users to generate code that competes with handwritten C programs on performance.

To measure the performance of code generated by Rupicola, we took a collection of tasks for which existing C implementations were available, implemented corresponding programs in Coq, and used Rupicola to compile them. Here, we give evidence that the performance of the resulting code is on par with C programs. To run these programs we do not use Bedrock2's compiler to RISC-V; instead we use a trivial pretty-printer to C to feed our programs to a regular C compiler (it would be possible to use Bedrock2's compiler or CompCert for greater assurance, albeit at a performance cost).

We chose programs from a variety of domains, including string manipulation, hash functions, and packet-manipulating (network) programs. Not discussed in the following are an additional suite of dozens of programs testing features around arithmetic, monadic extensions, and stack allocation (a subset of which are covered in table 5.1).

Table 5.2 gives a short description of each program that we benchmarked and fig. 5.1 shows the results of benchmarking (running on an Intel Core i7-4810MQ CPU @ 2.80GHz). As usual, benchmarks involving C compilers are very sensitive to small encoding decisions, so we measure performance across three compilers: overall the differences both in favor and against Rupicola are within the expected fluctuations across optimizing compilers, though we do suffer from a missed vectorization opportunity in upstr with GCC (section 5.2.1 discusses this result in more detail).

Comparing the performance of the original Coq code extracted to OCaml using Coq's native extraction features versus the C code produced by Rupicola yields results that are very problem-dependent: in most cases, extracting with Rupicola leads to algorithmic complexity changes (e.g. changing a linear nth lookup into a constant-time pointer dereference). When complexity is unchanged, a reasonable approximation is a speedup of 30 to 200× versus plain Coq extraction (typically closer to the latter). An example is given in fig. 5.2. It is possible to improve the performance of the OCaml code using potentially

Name	Source	Lemmas	Hints	End-to-End	Arithmetic	Inline	Arrays	Loops	Mutation
`crc32`	31	16	3		✓	✓	✓	✓	
Error-detecting code (cyclic redundancy check)									
`utf8`	56	-	6		✓	✓	✓		
Branchless UTF-8 decoding									
`m3s`	11	-	-		✓				
Scramble part of the Murmur3 algorithm									
`upstr`	21	-	6	✓	✓		✓	✓	✓
In-place string uppercase									
`ip`	37	3	7	✓	✓		✓	✓	
IP (one's-complement) checksum (RFC 1071)									
`fasta`	19	6	5	✓		✓	✓	✓	✓
In-place DNA sequence complement									
`fnv1a`	35	-	2		✓		✓	✓	
Fowler-Noll-Vo hash									

Table 5.2: Our benchmark suite. The "Source", "Lemmas", and "Hints" columns measure programmer effort in lines of code to write the original program and its signature, to prove the properties needed by Rupicola to compile that specific program, and to configure Rupicola prior to running the compiler, respectively. The "End-to-End" column indicates whether we have proofs from high-level specifications. The remaining columns describe which compiler extensions each program leverages.

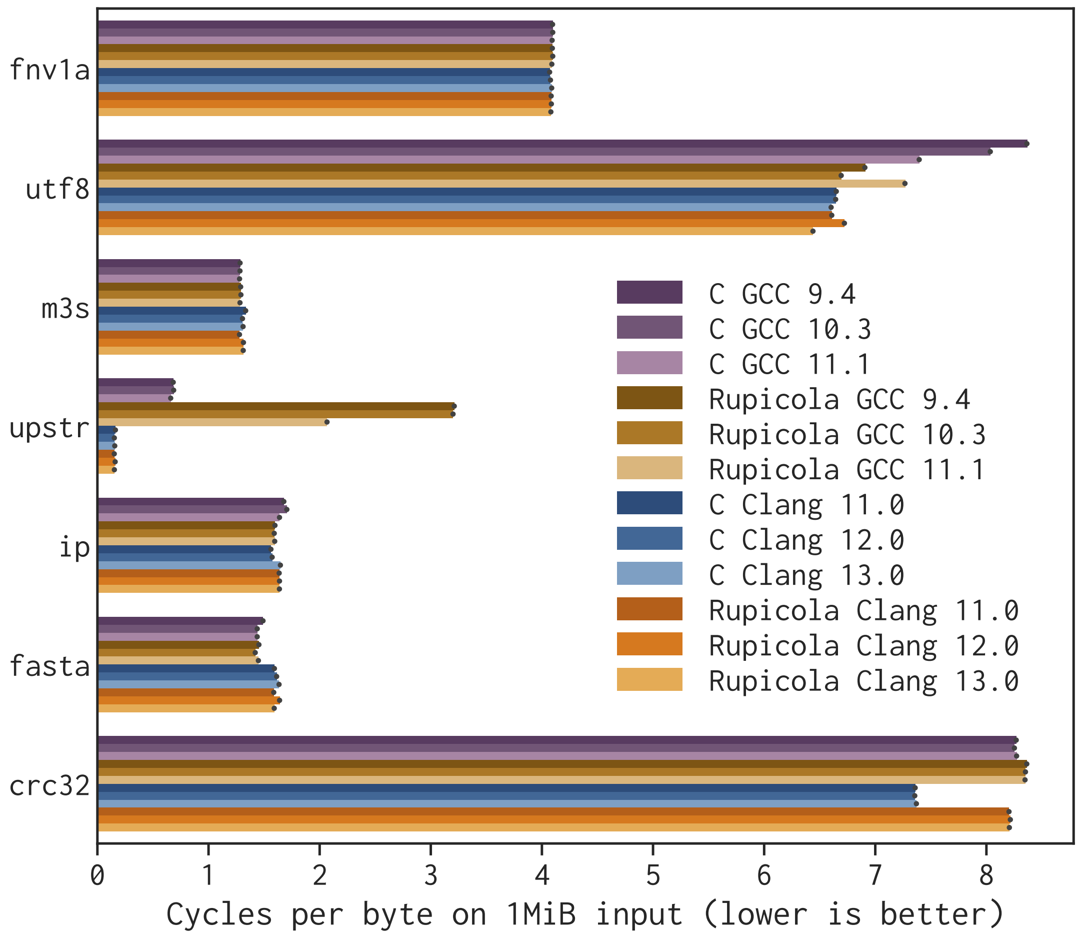
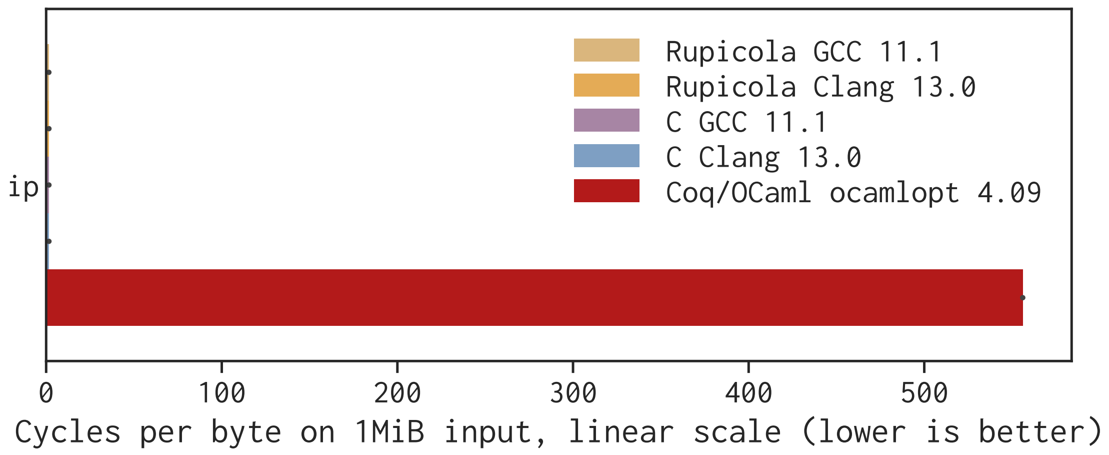


Figure 5.1: Performance benchmarks: Rupicola vs. handwritten C. Errors bars indicate 95% bootstrap confidence intervals over 1000 runs. The large fluctuations in upstr are due to inconsistent vectorization.



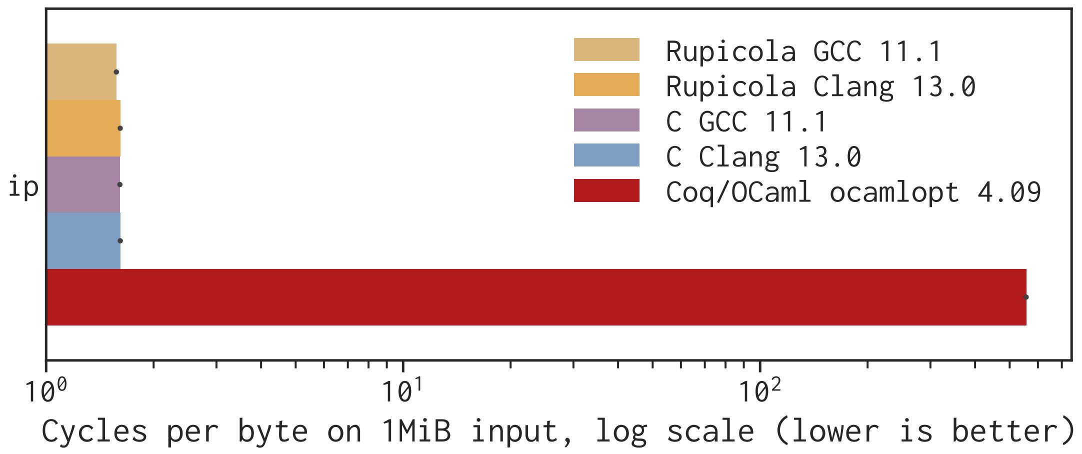


Figure 5.2: Performance benchmarks: Rupicola and handwritten C vs OCaml extracted from Coq specifications on one example. Error bars indicate 95% bootstrap confidence intervals over 1000 runs. The first plot uses a linear scale; the second one shows the same data with a logarithmic scale. The x axis on the second plot starts at 1 cycle/byte. Of four variants of the OCaml code generated by Coq (extracting the specs or the implementation, with or without additional extraction commands) only the fastest (specs with Coq integers unsoundly extracted to native ints) is shown, as the others have algorithmic complexity issues and hence do not complete in a reasonable amount of time.

unsound extraction commands (as shown in fig. 5.2, which maps Coq's unbounded integers to native integers), but only up to a point; and each additional customization of Coq's native extraction process is one opportunity for subtle bugs.

5.2.1 Performance considerations: C compilers and Bedrock2's C pretty-printer

Bedrock2 is a low-level language, but it is not equivalent to C: in some ways it is higher-level (in particular, its control flow is more structured), but in other ways it is much lower level: it has a single type, machine words, and all operations go through that type. Additionally, the semantics of Bedrock2 and (ISO) C differ: Bedrock2 has a very simple view of memory (a map from locations to bytes), whereas C attaches types to pointers and has complex restrictions on aliasing memory and accessing it through pointers of different types.

None of this is a concern when using the native (and verified) Bedrock2 toolchain ([14]). When pretty-printing Bedrock2 to C, however, these differences become problematic.

It might be possible to reconstruct enough information from a piece of Bedrock2 code to produce a richly typed C program, but this would require relatively complex reasoning and hence additional verification effort; instead, the authors of Bedrock2 chose to write a very simple translator that produces C code in which all variables have type uintptr_t, and all memory accesses go through a `memcpy` to avoid alignment issues.

The result is most likely not valid according to the ISO specification of C, a dubious distinction that Bedrock2's pretty-printer shares with lots of systems software: low-level C code tends to be written in the dialect of C supported by whichever compilers a project targets, not in ISO C[25]. I find it best to think of this C pretty-printing approach as a quick way to run programs compiled with Rupicola, rather than as a lasting piece of a trustworthy pipeline: in the long run, we will want to either verify the pretty-printer to C, pretty-print to a language with fewer pitfalls, or improve the Bedrock2 compiler enough to have competitive performance (hence my note in fig. 1.1 that the traditional compilation pass from Bedrock2 to ASM pass should be "ideally, verified").[26]

[25]This friction between C standards and real-world C programs, if undesirable, is at least not new. Back in 1988, Dennis Richie wrote the following of an early draft of a C standard: *The fundamental problem is that it is not possible to write real programs using the X3J11 definition of C. The committee has created an unreal language that no one can or will actually use.* [54]. For an overview of the debate, start with [67].

[26]Mapping Bedrock2's model to C is finicky business — early prototypes had type casting mistakes that led

5.2.1.1 Memory loads and stores

Modern C compilers are very good at optimizing these sort of programs, but we did have to go through a few iterations before we found a way to phrase memory accesses that compilers had no trouble with. The original implementation used the following (this discussion assumes a little-endian platform):

```c
static inline uintptr_t _br2_load(uintptr_t a, size_t sz) {
  uintptr_t r = 0;
  memcpy(&r, (void*)a, sz);
  return r;
}
```

Unfortunately, GCC 9 does not recognize the fact that memcopy will not *read* from pointer &r, and hence generates code that allocates space for r in memory and zeroes it out.

Explicitly masking the return value helps GCC (it recognizes that the mask is superfluous and optimizes it away, but additionally removes the unwanted store to &r), but this hurts performance in Clang 10, which does not eliminate the mask:

```c
static uintptr_t _br2_load(uintptr_t a, size_t sz) {
  uintptr_t r = 0;
  memcpy(&r, (void*)a, sizeof(uintptr_t));
  uintptr_t mask = (uintptr_t)-1 >> (8 * (sizeof(uintptr_t) - sz));
  return r & mask;
}
```

A commonly recommended alternative is to use byte-wide reads and use shifts and ors to reconstruct wider values [64]:

```c
#define READ8(S) ((255 & (S)[0]))
#define READ16LE(S) ((255 & (S)[1]) << 8 | (255 & (S)[0]))
#define READ32LE(S)                                             \
  ((uint32_t)(255 & (S)[3]) << 030 | (uint32_t)(255 & (S)[2]) << 020 | \
   (uint32_t)(255 & (S)[1]) << 010 | (uint32_t)(255 & (S)[0]) << 000)
#define READ64LE(S)                                             \
  ((uint64_t)(255 & (S)[7]) << 070 | (uint64_t)(255 & (S)[6]) << 060 | \
   (uint64_t)(255 & (S)[5]) << 050 | (uint64_t)(255 & (S)[4]) << 040 | \
```

to incorrect results and stack overflows. These thin, unverified layers of translation are a well-known source of issues in otherwise verified projects.

```
   (uint64_t)(255 & (S)[3]) << 030 | (uint64_t)(255 & (S)[2]) << 020 | \
   (uint64_t)(255 & (S)[1]) << 010 | (uint64_t)(255 & (S)[0]) << 000)

static inline uintptr_t _br2_load(uintptr_t a, size_t sz) {
  switch (sz) {
  case 1: return READ8((unsigned char*)a);
  case 2: return READ16LE((unsigned char*)a);
  case 4: return READ32LE((unsigned char*)a);
  case 8: return READ64LE((unsigned char*)a);
  default: __builtin_unreachable();
  }
}
```

Unfortunately, while both GCC 9+ and clang will optimize each branch of this function to a single memory load of the right width, that optimization interacts poorly with inlining in GCC, and as a result the inlined version of this function sometimes end up performing individual byte-wide loads when compiled with GCC 10. Eventually, we settled on the following definition, which is correctly optimized by GCC 9+ and Clang 10+:

```
static inline uintptr_t _br2_load(uintptr_t a, uintptr_t sz) {
  switch (sz) {
  case 1: { uint8_t  r = 0; memcpy(&r, (void*)a, 1); return r; }
  case 2: { uint16_t r = 0; memcpy(&r, (void*)a, 2); return r; }
  case 4: { uint32_t r = 0; memcpy(&r, (void*)a, 4); return r; }
  case 8: { uint64_t r = 0; memcpy(&r, (void*)a, 8); return r; }
  default: __builtin_unreachable();
  }
}
```

Naturally, none of this mess would be needed were we converting directly to a lower language than C.

5.2.1.2 Machine integers

Beyond the implementation of memory loads and stores, Bedrock2's use of uintptr_t when pretty-printing to C also introduces variations when compared to handwritten C code. This is particularly visible in the benchmark results for upstr, as code generated by Rupicola for the former is poorly optimized by GCC (but runs at speed comparable to the handwritten version in Clang).

That performance issue is readily explained: GCC simply misses a vectorization opportunity. It can be reproduced on the following simplified program:

```
#include <stdint.h>

void uintptr_mask(uintptr_t ubytes, int len) {
  for (int i = 0; i < len; i++)
    ((char*) ubytes)[i] = 0x5f & ((char*) ubytes)[i];
}
```

Compiling this code with GCC 9.4.0 `-O2 -ftree-vectorize -fopt-info-vec-all -S invert. -o /dev/null` produces the following output (GCC 10 and 11.1.0 produce similar output):

```
$ gcc-9 -O3 -fopt-info-vec-all -S uintptr_mask.c -o /dev/null
<stdin>:4:3: missed: couldn't vectorize loop
<stdin>:5:25: missed: not vectorized: compilation time alias: _4 = *_3;
*_3 = _5;
<stdin>:3:6: note: vectorized 0 loops in function.
```

This is in contrast to the following program, which GCC 9+ vectorizes successfully:

```
#include <stdint.h>

void uintptr_maskv(uintptr_t ubytes, int len) {
  char* bytes = (char*)ubytes;
  for (int i = 0; i < len; i++)
    bytes[i] = 0x5f & bytes[i];
}
```

```
$ gcc-9 -O3 -fopt-info-vec-all -S uintptr_maskv.c -o /dev/null
<stdin>:5:3: optimized: loop vectorized using 16 byte vectors
<stdin>:3:6: note: vectorized 1 loops in function.
```

Clang 10, in contrast, vectorizes both programs, and indeed we see no performance difference between the Rupicola version of upstr and the corresponding handwritten code in Clang:

```
$ clang-10 -O3 -Rpass=loop-vectorize -S uintptr_mask.c -o /dev/null
uintptr_mask.c:4:3: remark: vectorized loop↲
    (vectorization width: 16, interleaved count: 2)↲
    [-Rpass=loop-vectorize]
```

```
  for (int i = 0; i < len; i++)
  ^
$ clang-10 -O3 -Rpass=loop-vectorize -S uintptr_maskv.c -o /dev/null
uintptr_maskv.c:5:3: remark: vectorized loop↲
    (vectorization width: 16, interleaved count: 2)↲
    [-Rpass=loop-vectorize]
  for (int i = 0; i < len; i++)
  ^
```

5.2.1.3 Speed ups

The discussion above highlights one instance where a compiler struggles to optimize Rupicola's output as efficiently as it does handwritten code. The reverse also happens occasionally, but it is harder to make a systematic case for: any output that Rupicola produces could reasonably be produced, character-per-character, by hand, and then Rupicola would no longer have an edge — in that sense, a program produced by Rupicola can never really beat a handwritten program.

What I have observed, however, is cases where Rupicola's implementation of memory access plays better with compiler implementations. The original (handwritten) implementation of the `ip` benchmark, for instance, used `memcpy` to load potentially-unaligned 16-bits value from memory; the Rupicola implementation that I wrote at first, in contrast, used two separate 8-bit loads and then combined them (it was easier to write the code that way in Rupicola). It turns out that, for this particular example, Clang handles the latter better than the former, which led to a roughly 30% speedup when using Rupicola and Clang over handwritten C with Clang. Of course, the pattern used in Rupicola is also expressible in C, so the results in fig. 5.1 show the manually corrected handwritten code, which runs at the same speed as Rupicola.

6 Discussion

6.1 Compilation speed

The programs that Rupicola produces are fast, but Rupicola itself is not. On simple, straight-line code, it compiles in the order of tens of bindings per second. On programs that require more complex context manipulations, or on programs that need to invoke solvers to solve compilation side-conditions (e.g. programs with loops), it gets much slower.

Compiling the entirety of Rupicola's distribution on an 8-cores machine (dozens of example programs) takes in the order of minutes, and examples like the Utf8 decoder take tens of seconds to compile. The result is a compiler whose performance is sufficient for the programs that this thesis focuses on (and a few orders of magnitude faster than Fiat-to-Facade [44]!), but still painfully slow.

Where is all this time spent? Depending on the example, profiling suggests that about 50 and 80% of it is spent waiting for Coq's `autorewrite` tactic to realize than none of anywhere from 3 to 10 equations apply to the current goals[27] (the interesting part of Rupicola's work mostly happens in the `step_with_db` line in the profile shown below, which takes about 5% of the actual execution time):

```
 tactic                                 local  total   calls       max
───────────────────────────────────────┴──────┴──────┴───────┴─────────┘

─compile -------------------------------  0.0% 100.0%       1   38.922s
└compile_step --------------------------  0.0%  99.9%     229    2.501s
 ├─compile_autocleanup with (ident) ----  0.4%  56.1%     205    0.954s
 │└autorewrite with (ne_preident_list) ( 55.1%  55.2%     386    0.945s
 ├─compile_solve_side_conditions -------  38.0%  38.3%     333    1.546s
 │ ├─compile_autocleanup with (ident) --  0.3%  21.3%     145    1.182s
 │ │└autorewrite with (ne_preident_list) 20.5%  20.5%     207    1.173s
```

[27] Performance issues with `autorewrite` are well known, and generally Rupicola is not running into particularly exciting Coq performance issues — just run-of-the-mill slowness.

```
 │ ├─solve_map_get_goal ----------------  0.0%  10.4%      12    1.539s
 │ │└solve_map_get_goal_step -----------  4.4%  10.4%      33    1.529s
 │ │ ├─solve_map_get_goal_refl ---------  0.0%   6.0%       3    1.523s
 │ │ │└reify_map -----------------------  0.2%   5.6%       3    1.470s
 │ │ │└set_change (uconstr) with (uconst  0.0%   5.3%       3    1.433s
 │ │ │└set (sx := x) -------------------  5.0%   5.0%       3    1.393s
 │ │ └─rewrite map.get_put_diff --------  4.2%   4.2%       9    0.210s
 │ └─step_with_db (ident) --------------  0.0%   5.7%      62    0.684s
 │  └unshelve (tactic1) ----------------  0.0%   5.7%     103    0.684s
 │  └typeclasses eauto (nat_or_var_opt)   0.1%   5.7%     103    0.684s
 │  └cbn -------------------------------  4.3%   4.3%      57    0.609s
 └─compile_triple ----------------------  0.0%   5.2%      24    1.215s
  └compile_unset_and_skip --------------  0.0%   3.1%       5    1.209s
  └compile_cleanup_post ----------------  0.2%   3.1%       7    0.250s
  └compile_autocleanup with (ident) ----  0.1%   2.7%      16    0.210s
  └autorewrite with (ne_preident_list) (  2.6%   2.6%      30    0.203s
```

These profiles also do not suggest that there would be much to be gained in migrating from Coq's venerable Ltac1 tactic language to Ltac2. It is possible to speed things up quite a bit by being more discerning in applying autorewrite, but this adds complexity for users, who would have to grasp a more complex set of hint databases. Instead, I have chosen to go for maximal ease-of-use.

Performance issues like the ones that Rupicola suffers from are a common plague of projects that rely heavily on Coq's proof language. They stem primarily from a mix of asymptotically suboptimal algorithms and unoptimized implementation, which are not particularly surprising (Coq was not originally designed with this kind of heavy automation in mind).

Other performance issues pop up from time to time that come from unexpected pitfalls, often related to reduction: to give two brief examples, at one point a colleague tracked down the source of a tenfold slowdown to Coq eagerly unfolding the `noskip` function described above while checking a proof[28]; at another I noticed that each unfolding hint added to a Rupicola database was causing an exponential increase in proof time[29] (there is, thankfully, a workaround for the problem).

[28] Rupicola commit 30dbd80d761ad3e8a26dd19d3cdcc264249ffd45.

[29] Coq bug #14874.

6.2 Incremental compiler construction and backtracking

Rupicola's design makes it easy to build compilers incrementally. In that context, and important feature is the ability do debug incomplete compilation runs: users need to be able to straightforwardly determine which construct Rupicola failed to compile, and how to add support for it.

Let us look at an example to see what concretely happens when Rupicola does not know how to compile a source code construct. We look at a simple program that takes as input an array of bytes, reinterprets it as an array of little-endian machine words, and counts the number of matches for a given search term in the resulting array.

```
Definition count_ws (data: ListArray.t byte) (needle: word) :=
  let/n r := 0 in
  let/n data := bs2ws data in
  let/n r := ListArray.fold_left (fun r w64 ⇒
       let/n hit := word.eqb w64 needle in
       let/n r := r + Z.b2z hit in
       r)
    data 0 in
  let/n data := ws2bs data in
  r.
```

Instead of applying the mask byte by byte, the program starts by casting its input into a list of 64-bit words, processes these words, and finally casts the result back to a lists of bytes. Without loading appropriate libraries, Rupicola will not recognize these patterns:

```
Instance spec_of_count_ws : spec_of "count_ws" :=
  fnspec! "count_ws" ptr wlen needle / (bs: list byte) R ~> r, {
    requires tr mem :=
      wlen = of_nat (length bs) ∧
      Z.of_nat (length bs) < 2 ^ width ∧
      (Datatypes.length bs mod bytes_per_word = 0)%nat ∧
      (bytes ptr bs ⋆ R) mem;
    ensures tr' mem' :=
      tr' = tr ∧ r = of_Z (count_ws bs needle) ∧
      (bytes ptr bs ⋆ R) mem'
  }.

Context (bytes_per_word_nz : bytes_per_word ≠ 0%nat).
Context (bytes_per_word_range: 0 < Z.of_nat bytes_per_word < 2 ^ width).
```

```
Derive count_ws_br2fn SuchThat
  (defn! "count_ws"("data", "len", "needle") ~> "r" { count_ws_br2fn },
   implements count_ws) As count_ws_br2fn_ok.
Proof.
  compile.
```

```
Compilation incomplete.

You may need to add new compilation lemmas using `Hint Extern 1 => simple eapply … : compiler` or to t
```

```
________________________________________________________________________
{{ tr; mem0;
   #{ "data" => ptr; "len" => of_Z Z.of_nat (Datatypes.length bs);
      "needle" => needle; "r" => of_Z v }#; functions }}
  ?k_impl
{{ pred
     (let/n data as "data" := bs2ws bs in
      let/n r as "r" :=
        ListArray.fold_left
          (fun (r : Z) (w64 : word) =>
           let/n hit as "hit" := word.eqb w64 needle in
           let/n r0 as "r" := r + Z.b2z hit in r0) data 0 in
      let/n _ as "data" := ws2bs data in r) }}
```

In fact, compilation stops immediately, and Rupicola prints a message indicating that we need new lemmas. Inspecting the goal indicates that Rupicola already combined the binding for count and then ran into trouble with the second binding (the call to bs2ws), so we add a lemma for it:

```
Lemma compile_bs2ws [t m l σ] (bs: list byte):
  let v := bs2ws bs in

  forall {P} {pred: P v -> _} {k: nlet_eq_k P v} K r bs_var ptr,

    (bytes ptr bs ⋆ r) m ->
    (length bs mod bytes_per_word = 0)%nat ->

    (forall m',
     (words ptr v ⋆ r) m' ->
     {{ t; m'; l; σ }} K {{ pred (k v eq_refl) }}) ->

    {{ t; m; l; σ }} K {{ pred (nlet_eq [bs_var] v k) }}.
Proof. intros; seprewrite_in bytes_as_words H; eauto. Qed.
```

One we have defined this new lemma, we can plug it in an make a bit more progress:

```
Hint Extern 1 => simple eapply compile_bs2ws; shelve : compiler.
```

```
compile_step; try solve [repeat compile_step].
```

```
────────────────────────────────────────────────────────────────
{{ tr; m';
   #{ "data" => ptr; "len" => of_Z Z.of_nat (Datatypes.length bs);
      "needle" => needle; "r" => of_Z v }#; functions }}
  ?k_impl
{{ pred
     (let/n r as "r" :=
        ListArray.fold_left
          (fun (r : Z) (w64 : word) =>
           let/n hit as "hit" := word.eqb w64 needle in
           let/n r0 as "r" := r + Z.b2z hit in r0)
          (bs2ws bs) 0 in
      let/n _ as "data" := ws2bs (bs2ws bs) in r) }}
```

This time Rupicola stops because we have not given it a way to compile `ListArray.fold_left`; the default is fine, so we import the `UnsizedListArrayCompiler` module:

```
Import UnsizedListArrayCompiler.
compile_step; try solve [repeat compile_step].
```

```
────────────────────────────────────────────────────────────────
0 ≤ Z.of_nat (Datatypes.length (bs2ws bs)) < 2 ^ width
```

```
────────────────────────────────────────────────────────────────
DEXPR m'
  #{ "data" => ptr; "len" => of_Z Z.of_nat (Datatypes.length bs);
     "needle" => needle; "r" => of_Z v; (gs "_gs_from") => of_Z 0 }#
  ?e (of_Z (Z.of_nat (Datatypes.length (bs2ws bs))))
```

```
────────────────────────────────────────────────────────────────
{{ tr; mem1;
   #{ "data" => ptr; "len" => of_Z Z.of_nat (Datatypes.length bs);
      "needle" => needle; "r" => of_Z v0;
      (gs "_gs_from") => of_Z Z.of_nat (Datatypes.length (bs2ws bs));
      (gs "_gs_to") => of_Z Z.of_nat (Datatypes.length (bs2ws bs)) }#; functions }}
  ?c
{{ pred (let/n _ as "data" := ws2bs (bs2ws bs) in v0) }}
```

This time we see that we are missing three components: a side condition about the length of the result of converting bytes to words; an expression compilation goal to compute that length for the loop; and a final goal to cast the list of words back to a list of bytes. We can make progress on the first two by registering a hint:

```
    Hint Rewrite bs2ws_len using solve[eauto] : compiler_side_conditions.
    Hint Rewrite Nat2Z_inj_div : compiler_side_conditions.
    Hint Extern 10 => nia : compiler_side_conditions.

    all: repeat compile_step.
```

```
{{ tr; mem1;
   #{ "data" => ptr; "len" => of_Z Z.of_nat (Datatypes.length bs);
      "needle" => needle; "r" => of_Z v0;
      (gs "_gs_from") => of_Z
                             Z.of_nat (Datatypes.length bs) /
                             Z.of_nat bytes_per_word;
      (gs "_gs_to") => of_Z
                           Z.of_nat (Datatypes.length bs) / Z.of_nat bytes_per_word }#;
   functions }}
  ?Goal
{{ pred (let/n _ as "data" := ws2bs (bs2ws bs) in v0) }}
```

And finally all that is left is the call that casts the data back, which requires a lemma very similar to the previous one:

```
    Hint Extern 1 => simple eapply compile_bs2ws_rev; shelve : compiler.
    repeat repeat compile_step.
  Qed.
```

Once this process of interactive development is complete, we can move our new lemmas out of the proof (and possibly into a library, if they are — like here — generally applicable). The five hints that make up the proof are our compiler.

This interactive debugging and compiler construction experience is one of the main reasons why backtracking is so unappealing in the context of Rupicola[30]: as long as there is no backtracking, the user can be confident that the compilation goal they are looking at is the furthest that Rupicola could progress. In a compiler with backtracking, the user instead has to start by reconstructing the failed compilation path, and then understand why that path failed (this is a common problem with Coq's eauto tactic, and in fact with many automated theorem proving technologies: eauto is very pleasant to use when it works, but when a call to it fails, e.g. due to a change in a theorem statement, the debugging experience is quite poor).

[30]As of this writing backtracking is used in only one place in Rupicola, for deciding between signed and unsigned comparison operators when compiling arithmetic expressions on integers.

Another important reason for avoiding backtracking is predictability: we want users to be in complete control of the code generation process, so to have at most one lemma applicable to each source code pattern loaded at any time — it is the responsibility of the user to annotate their code to indicate which pattern applies, and to load the appropriate compilation modules.

Finally, the last reason to avoiding backtracking is performance: going down unsuccessful compilation paths wastes time.

6.3 Nondeterminism

The predecessor to Rupicola, Fiat-to-Facade, hardcoded the nondeterminism monad: it inherited that trait from the Fiat system [11], and we initially worried that Rupicola would not have the same flexibility.

How Rupicola deals with non-determinism depends on its kind:

6.3.1 Erasable nondeterminism

Some datastructures expose a deterministic interface while relying on nondeterminism internally. A fixed-size stack, for example, contains a data section and some uninitialized space to grow into. Methods of the stack do not provide access to the unitialized section, so the stack exposes a deterministic interface.

Other structures operate deterministically, but we may prefer to abstract certain details of their implementation. For example, a binary search tree used to implement a set datastructure with `insert`, `remove`, and `contains` methods will answer `contains` queries deterministically, even if its exact layout is unknown (e.g. we may not know which element is at the root).

In these cases, it is possible to work with deterministic models of the data structure and to capture the nondeterminism at the separation-logic predicate level. Specifically, we can model both the stack and the tree as the list of elements that they contain, and hide the nondeterminism using existential quantification within our representation predicates. For stacks, we might write the following, which is essentially what we do with the buffer API of section 4.6.2.2:

```
Definition stack_at addr capacity model :=
    fun m: mem ⇒ ∃ suffix,
      length (model ++ suffix) = capacity ∧
      bytes addr (model ++ suffix) m.
```

For sets, we might write something similar to the following:

```
Definition set_at {E} addr element_at (model: list E) :=
  fun m ⇒ ∃ t: tree E,
    ∃ t: tree E,
      is_bst t ∧
      is_permutation model (tree_elements t) ∧
      tree_at addr element_at t m.
```

6.3.2 Observable nondeterminism

Other datastructures expose non-determinism to their callers.

This may be because the structure actually implements nondeterministic operations (maybe because one of its operations uses concurrent programming under the hood), or it may be because the model that we chose omits details of the implementation (perhaps to allow changes in the implementation).

For example, if we were to add a `peek` operation to our binary search tree returning the root of the tree, we would get different results depending on the layout of the tree, which did not matter for `contains` tests.

Nondeterminism stemming from underspecification is common, but not excessively so: indeed, if we want to be able to verify a low-level program that implements the operation, we need a sufficiently precise representation invariant: an invariant that abstracts away the hash function used to build a hash table by existentially quantifying it, for example, would not permit us to prove the correctness of a `lookup` operation.

When nondeterminism or underspecification is present, we write Rupicola programs in the non-determinism monad, and we adjust representation predicates and function postconditions accordingly. For separation logic predicates, we can generalize any deterministic predicate `element_at` over a family of possible objects:

```
Definition nondet_at {A} addr
           (element_at: word → A → mem → Prop)
           (val: A → Prop) mem :=
  ∃ a, val a ∧ element_at addr a mem.
```

For function postconditions, where we would have previously asserted that the output of a Bedrock2 function should equal a given Gallina value, we assert instead that the value returned by the Bedrock2 program should belong to the set of values allowed by the nondeterministic Gallina program (section 4.5.1.2).

6.4 Trusted base

Which moving parts does one have to trust when running a program compiled with Rupicola? Roughly the following:

Rupicola's inputs, or any higher-level specifications Rupicola proves that its outputs match its inputs, but no more — bugs in latter are dutifully replicated in the former. If Rupicola's inputs are verified against — or generated from — higher-level specifications, then these need to be trusted. In the example of the IP checksum code, this means trusting the high-level Gallina implementation.

Bedrock2's pretty-printer to C — or to RISCV The conversion of Rupicola's outputs from Bedrock2 to C is done using a small (~200 lines) but unverified pretty-printer written in Gallina. When compiling using Bedrock, the trusted base is reduced to the Coq notations that are used to print a byte dump of the assembled RISCV code.

Any unverified portions of the lower-level compilation toolchain Going through C to compile and run Rupicola requires trusting the lower-level compilation toolchain: compiler, linker, and assembler. This is not an issue when compiling with Bedrock2's verified compiler.

Coq's proof checker Rupicola's guarantees are only as good as those provided by the Coq proof assistant: bugs in its proof checker could lead it to accept incorrect proofs.

The environment in which programs execute, and assumptions about it Issues in the operating system, if any, or hardware components of the machine running programs compiled with Rupicola can derail an otherwise-verified execution.

6.5 Contributions and credits

I have been lucky to collaborate with a wonderful group of people at MIT; parts of Rupicola's development are due to them.

First and foremost is Jade Philipoom. We brainstormed many aspects of the design of Rupicola together early on, pair-programmed some of the core definitions, and she contributed multiple of Rupicola's early examples and library functions, especially cryptography-related ones. Jade and I also spent a lot of time brainstorming ways to support complex borrowing and mutation schemes that are not part of this thesis.

Dustin Jamner later worked on the implementation of Rupicola's support for inline tables and stack allocation.

Andres Erbsen wrote and ran most of Rupicola's benchmarks.

A rough count attributing each source code line to the last author having edited them (`git blame`) in the current head of Rupicola's git repository, excluding a large Bedrock2-related refactoring, returns the following results:

```
  741 author Dustin Jamner
 2759 author Jade Philipoom
15930 author Clément Pit-Claudel
```

Outside of Rupicola's repository, Ashley Lin is compiling modular exponentiation routines, and I am working with Dustin Jamner and Andres Erbsen to use Rupicola to derive implementations of cryptographic primitives.

Parts of this dissertation are under review for publication as a conference paper co-authored with Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala.

7 Related work

Generally speaking, Rupicola's unique strength is its combination of extensibility, foundational proofs, and high performance. Specifically, Rupicola supports extension of the source to introduce new monads and datatypes in the input; extension of the compilation process to support new data representations and foreign function calls; support for soundly linking against code written directly in Bedrock2; explicit memory management without automatic garbage collection; and the ability to derive end-to-end proofs, as well as the embedding in a proof assistant that makes the whole system interactive. Most other work lacks at least one of these aspects.

7.1 Binary code extraction and extensible extraction of imperative code

Maybe the most closely relevant competing project in the space of relational compilation is Fiat-to-Facade (F2F) [48, 44]. It was the first demonstration of an end-to-end pipeline for deriving code automatically from high-level specifications to low-level code, and it strove for both performance and extensibility in a foundational context. Unfortunately, it did not fully realize these goals:

- Extensions are possible in F2F, but many are hard to express, owing to the choice of a linear language as the compilation target. (consequently, authors are encouraged to write most complex code directly in the lower-level languages, leaving F2F for the glue code surrounding calls to low-level functions). Rupicola uses separation logic directly, and some of the data-structure primitives in Rupicola's standard library are in fact derived using Rupicola.
- F2F proves partial correctness: programs in F2F are only correct up to termination. Rupicola proofs are total.

- F2F's code-derivation strategy is based on setoid rewriting, leading to significant performance issues (a typical compilation run may take tens of minutes even for a small program, compared to just tens of seconds for Rupicola).
- F2F compilers are assembled from compilation lemmas but cannot then be extended: extensibility is done through hooks instead of hint databases as in Rupicola (the latter is made possible, or at least much easier, by Rupicola's encoding of continuations).
- F2F hardcodes the nondeterminism monad and does not support extensions to other monads. Rupicola encodes its pre- and postconditions in a different way that is agnostic to the ambient monad, making it easy to represent values with custom representations and to compile computations that represent target-language effects using a custom monad.

The line of work on Imperative/HOL by P. Lammich et. al. is also closely related to this work, starting from extraction from Isabelle/HOL to Imperative/HOL in 2015 [23] (a refinement framework to translate functional code into a shallowly embedded imperative language with GC), all the way to the generation of verified LLVM from Isabelle/HOL in 2019 [24] through a similar refinement process. Early work targeted a shallowly embedded language, but the latest work extracts directly to LLVM. The main difference is the scope of the translation: LLVM/HOL uses a direct embedding of LLVM into HOL, so relational compilation is used to perform what is essentially a one-to-one translation where all effects in the source are encoded extensionally. Rupicola, on the other hand, accepts more complex inputs and supports most effects intensionally, adding complexity in the compiler in exchange for a richer input language. (Another difference is that Rupicola integrates into a verified pipeline: code extracted with Rupicola can be soundly compiled and linked against other code written in Bedrock2 or directly in machine code, within Coq, whereas there is no verified implementation of LLVM is Isabelle/HOL today.)

Another closely related line of work uses proof-producing extraction to translate HOL programs to deeply embedded CakeML (a dialect of ML for which there exists a verified compiler [22]) [35, 36, 17]. It bridges a much narrower gap than Rupicola does (it targets a language with garbage collection), but in exchange it offers a much more complete translation pipeline, in the sense that it supports a better-defined and larger part of its input language, HOL.

Members of the F* team take a slightly different approach in KreMLin [52], an extraction framework from Low* (an imperative subset of F* to C: the extraction process is not verified (though a proof-producing strategy for a subset was described in [32]), but the close match between the Low* style and C means that KreMLin's trusted base is reasonably small, and the emphasis on output-code quality means that C code from KreMLin can easily be integrated into larger, potentially unverified C programs (as has in fact been done with cryptography routines [51]). This strategy is viable because F* provides convenient facilities for reasoning about stateful programs in shallowly embedded style, making it possible to prove the connection between code written in high-level functional style and low-level imperative style without resorting to reasoning about deeply embedded low-level terms. More recent developments explore metaprogramming and code generation using stateful functors [50].

The authors of Cogent [38] take a different approach. Instead of translating between two languages (one functional and one imperative), they guarantee (using a restrictive type system) that all Cogent programs admit an efficient implementation that does not depend on a runtime or a garbage collector (this is similar to the way in which Facade [48] was essentially a linear type system on top of Cito [65]). As a result, unlike Rupicola and F2F, Cogent is complete, but it is also much more restrictive: unlike Rupicola, it does not support arbitrary user-supplied extensions, nor custom translation of specific high-level patterns: all optimizations must be expressed in the source program, not as transformations to be applied as part of the source-to-target translation process.

7.2 Other related work

7.2.1 Coq extraction

Coq's traditional extraction mechanism [41, 42] is not machine-verified, but it is proven on paper [28], and it supports a form of (unsound) extension by remapping constructors and functions to arbitrary OCaml expressions, a feature very commonly used in large extracted Coq developments. With sufficiently arcane combinations of extraction commands, it is often possible to improve performance significantly, at some risk to soundness. More principled are approaches based on reification: with a sufficiently restricted subset of Gallina, it is possible to reify terms into a deeply embedded AST using Ltac's reflection and certify correctness of that translation by interpreting deeply embedded results back into

Gallina [34, 68]. This approach works best when the input language is restricted in a way that makes the interpretation function easy to write.

The CertiCoq project [1] supports verified compilation from Coq to assembly: it starts by reifying all of Coq into a deeply embedded AST type (this step is not proof-producing) and then proceeds as a traditional verified compiler. Unlike in Rupicola, the extraction process is not extensible, which forces users to pay the price of inefficiencies at the Gallina level — but in exchange it supports all of Coq instead of small, custom subsets of it.

7.2.2 Verified compilation

More generally, the last few years have seen an explosion of work on the topic of compiler verification, most notably with CakeML [22] and CompCert [27]. F2F depended on a verified compiler called Cito [65]; Rupicola uses Bedrock2 [14].

7.2.3 Translation validation

Complete verification of a compiler can be onerous, and verifying that the compiler produces correct outputs on *all* inputs is often qualitatively more complex that establishing that property for any given input/output pair. As a result, many verified systems employ *translation validation* instead of verification: a (trustworthy, ideally verified) checker is used to confirm, for each run of the compiler, that the outputs are correct. The problem is undecidable for most input and output languages, so a variety of heuristics coexist in the literature [49, 37, 62, 63, 61, 20, 66], some quite close to the relational extraction style that I advocate [16]. often, the unverified compiler produces additional information ("witnesses") besides the output program, and that information is used to facilitate the work of the checker. It would not be unreasonable to classify Rupicola as a translation-validation system, since it uses unverified Ltac scripts to generate output programs along with "witnesses" of correctness in the form of Coq proof terms. That proof term is then validated by Coq's kernel before being accepted, but this "verification" step is trivial, in the sense that it can only fail in the presence of bugs in the implementation of the Ltac scripting language and its primitive tactics (and not because of bugs in compilers built with Rupicola, since these bugs manifest as compilation stopping early, in a partially compiled state).

There is an additional sense in which Rupicola is a translation-validation system, however: at its core, Rupicola is really a strongest postcondition (SP) calculator for Bedrock2 code, except the code that it evaluates is constructed piecemeal, based on the shape of a Gallina value (see Box 1). Since evaluation in Gallina (unfolding consecutive let bindings) and partial evaluation in Bedrock2 (SP computation) proceed in lockstep as the Bedrock2 code is being derived, all that typically remains to be done at the end of a Rupicola compilation run is to unify the result of symbolic evaluation with the expected postcondition — the "translation-validation" step. This step is simple because each individual compilation lemma maintains the connection between Bedrock2 and Gallina.

7.2.4 Automatic generation of functional models (importing code into Coq)

The typical way to run code written in an interactive theorem prover like Coq [58] or in a verification-aware language like Dafny [25] is to use program extraction to generate executable code. This may be undesirable when precise control over the final software artifact is needed, e.g. for performance or ease-of-integration reasons (for example, Dafny supports code generation in Go, but the result is neither fully idiomatic nor customizable). In these cases, an alternative to extraction (which generates executable code from functional models) is to generate functional models from executable code, i.e. to take a handwritten program and automatically translate it into an equivalent model in Coq, Dafny, or some other tool. This is how hs-to-coq [55], CFML [8], or Goose [6] operate: they take code in Haskell, OCaml, and Go respectively and translate it into a Gallina model that can be reasoned about using Coq.

7.2.5 Optimizing compilation of functional programs

Finally, there is of course a long history of developing optimizations for functional languages. Recent development of note are the development of the FLambda backend for the OCaml compiler, which significantly reduces boxing [7]; the introduction of a clever analysis in the Lean compiler to automatically reuse space (introducing a form of transparent mutation), leveraging reference counts to determine when it is safe to overwrite a value [53]; and multiple optimizations for CertiCoq [29, 39].

7.2.6 Extensible compilers, optimization DSLs, and program synthesis

Stepping further, Rupicola's design shares a lot with work on extensible compilation and domain-specific languages for optimization [40, 60, 57, 26] and more generally with work on automated program derivation and program synthesis, all the way back to deductive program synthesis [31].

8 Conclusion

This thesis introduced the unifying framework of *relational compilation* and presented Rupicola, a relational-compilation toolkit that leverages modular compiler extensions to derive high-performance, verified low-level programs automatically from functional sources. Rupicola is unique in its combination of extensibility, foundational proofs, and performance. We are in the process of extending it to support further application domains, and we are looking into integrating its verified outputs into existing widely used libraries. I hope that, in the long run, the techniques presented in this thesis will provide a solid foundation for systems verified from end to end and worthy of trust.

9 Bibliography

[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A Verified Compiler for Coq. In *CoqPL'17: The Third International Workshop on Coq for PL*.

[2] Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbrücken, Germany) *(ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 1–17.

[3] David Aspinall. 2000. Proof General: A Generic Tool for Proof Development. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000*. 38–42. https://doi.org/10.1007/3-540-46419-0_3

[4] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*. 207–228. https://doi.org/10.1007/11745853_14

[5] Edwin C. Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[6] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. 243–258. https://doi.org/10.1145/3341301.3359632

[7] Pierre Chambart, Mark Shinwell, Damien Doligez, and OCaml Contributors. 2016. *Optimization with FLambda*. https://ocaml.org/manual/flambda.html

[8] Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 418–430. https://doi.org/10.1145/2034773.2034828

[9] Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. 2017. The End of History? Using a Proof Assistant to Replace Language Design with Library Design. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:15. https://doi.org/10.4230/LIPIcs.SNAPL.2017.3

[10] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[11] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15*. ACM Press, 689–700. https://doi.org/10.1145/2676726.2677006

[12] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-by-construction Derivation of Decoders and Encoders from Binary Formats. *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 82 (July 2019), 29 pages. https://doi.org/10.1145/3341686

[13] Andres Erbsen. 2017. *Crafting certified elliptic curve cryptography implementations in Coq*. Master's thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/112843

[14] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 604–619. https://doi.org/10.1145/3453483.3454065

[15] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala.

2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. *2019 IEEE Symposium on Security and Privacy (SP)* (May 2019). `https://doi.org/10.1109/sp.2019.00005`

[16] Yannick Forster and Fabian Kunze. 2019. A certifying extraction with time bounds from Coq to call-by-value λ-calculus. In *Interactive Theorem Proving - 10th International Conference, ITP 2019, Portland, USA*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 17:1–17:19. Also available as arXiv:1904.11818.

[17] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. 2018. Proof-Producing Synthesis of CakeML with I/O and Local State from Monadic HOL Functions. *Lecture Notes in Computer Science* (2018), 646–662. `https://doi.org/10.1007/978-3-319-94205-6_42`

[18] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. 2007. A history of Haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*. 1–55. `https://doi.org/10.1145/1238844.1238856`

[19] Lars Hupel and Tobias Nipkow. 2018. A Verified Compiler from Isabelle/HOL to CakeML. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 999–1026. `https://doi.org/10.1007/978-3-319-89884-1_35`

[20] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 397–416.

[21] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. 2018. Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB - (Short Paper). In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10895)*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer, 362–369. `https://doi.org/10.1007/978-3-319-94821-8_21`

[22] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML:

A Verified Implementation of ML. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, January 20-21, 2014*. 179–192. https://doi.org/10.1145/2535838.2535841

[23] Peter Lammich. 2015. Refinement to Imperative/HOL. In *International Conference on Interactive Theorem Proving, ITP 2015, Nanjing, China, August 24-27, 2015*. 253–269. https://doi.org/10.1007/978-3-319-22102-1_17

[24] Peter Lammich. 2019. Generating Verified LLVM from Isabelle/HOL. In *International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*. 22:1–22:19. https://doi.org/10.4230/LIPIcs.ITP.2019.22

[25] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. *Logic for Programming, Artificial Intelligence, and Reasoning* (2010), 348–370.

[26] Sorin Lerner, Todd D. Millstein, Erika Rice, and Craig Chambers. 2005. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. 364–377. https://doi.org/10.1145/1040305.1040335

[27] Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 42–54. https://doi.org/10.1145/1111037.1111042

[28] Pierre Letouzey. 2002. A New Extraction for Coq, In International Workshop on Types for Proofs and Programs, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002. *Types for Proofs and Programs*, 200–219. https://doi.org/10.1007/3-540-39185-1_12

[29] John M. Li and Andrew W. Appel. 2021. Deriving Efficient Program Transformations from Rewrite Rules. *Proc. ACM Program. Lang.* 5, ICFP, Article 74 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473579

[30] Andreas Lööw and Magnus O. Myreen. 2019. A proof-producing translator for Verilog development in HOL. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. 99–108. https://doi.org/10.1109/FormaliSE.2019.00020

[31] Zohar Manna and Richard J. Waldinger. 1980. A Deductive Approach to Program

Synthesis. *ACM Transactions on Programming Languages and Systems* 2, 1 (Jan. 1980), 90–121. https://doi.org/10.1145/357084.357090

[32] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, Luís Caires (Ed.). Springer International Publishing, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2

[33] Peter L. Montgomery. 1987. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.* 48, 177 (1987), 243–264. https://doi.org/10.1090/s0025-5718-1987-0866113-7

[34] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: Minimizing the Coq Extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 172–185. https://doi.org/10.1145/3167089

[35] Magnus O. Myreen and Scott Owens. 2012. Proof-producing Synthesis of ML from Higher-order Logic. In *ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, Copenhagen, Denmark, September 9-15, 2012*. 115–126. https://doi.org/10.1145/2364527.2364545

[36] Magnus O. Myreen and Scott Owens. 2014. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming* 24, 2-3 (Jan 2014), 284–315. https://doi.org/10.1017/s0956796813000282

[37] George C. Necula. 2000. Translation validation for an optimizing compiler. *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation - PLDI '00* (2000). https://doi.org/10.1145/349299.349314

[38] Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming* 31 (2021), 25. https://doi.org/10.1017/S095679682100023X

[39] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. 2021. Compositional Optimizations for CertiCoq. *Proc. ACM Program. Lang.* 5, ICFP, Article 86 (Aug. 2021), 30 pages. https://doi.org/10.1145/3473591

[40] Lionel Parreaux. 2020. *Type-Safe Metaprogramming and Compilation Techniques For Designing Efficient Systems in High-Level Languages*. Ph.D. Dissertation. EPFL, Lausanne. https://doi.org/10.5075/epfl-thesis-10285

[41] Christine Paulin-Mohring. 1989. *Extraction de programmes dans le Calcul des Constructions*. Theses. Université Paris-Diderot - Paris VII. https://tel.archives-ouvertes.fr/tel-00431825

[42] Christine Paulin-Mohring and Benjamin Werner. 1993. Synthesis of ML Programs in the System Coq. *J. Symb. Comput.* 15, 5–6 (May 1993), 607–640. https://doi.org/10.1016/S0747-7171(06)80007-6

[43] Jade Philipoom. 2018. *Correct-by-construction finite field arithmetic in Coq*. Master's thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/119582

[44] Clément Pit-Claudel. 2016. *Compilation Using Correct-by-Construction Program Synthesis*. Master's thesis. Massachusetts Institute of Technology. http://pit-claudel.fr/clement/MSc/

[45] Clément Pit-Claudel. 2020. Untangling Mechanized Proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (Virtual, USA) *(SLE 2020)*. Association for Computing Machinery, New York, NY, USA, 155–174. https://doi.org/10.1145/3426425.3426940

[46] Clément Pit-Claudel and Thomas Bourgeat. 2021. An experience report on writing usable DSLs in Coq. In *CoqPL'21: The Seventh International Workshop on Coq for PL*. https://pit-claudel.fr/clement/papers/koika-dsl-CoqPL21.pdf

[47] Clément Pit-Claudel and Pierre Courtieu. 2016. Company-Coq: Taking Proof General one step closer to a real IDE. In *CoqPL'16: The Second International Workshop on Coq for PL*. https://doi.org/10.5281/zenodo.44331

[48] Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *Automated Reasoning: 10th*

International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12167), Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, 119–137. https://doi.org/10.1007/978-3-030-51054-1_7

[49] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems - 4th International Conference, TACAS'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28 – April 4, 1998 Proceedings*, Bernhard Steffen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166.

[50] Jonathan Protzenko and Son Ho. 2021. Zero-cost meta-programmed stateful functors in F. *CoRR* abs/2102.01644 (2021). arXiv:2102.01644 https://arxiv.org/abs/2102.01644

[51] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cedric Fournet, and et al. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. *2020 IEEE Symposium on Security and Privacy (SP)* (May 2020). https://doi.org/10.1109/sp40000.2020.00114

[52] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-level Programming Embedded in F*. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 17:1–17:29. https://doi.org/10.1145/3110261

[53] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage free reference counting with reuse. *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Jun 2021). https://doi.org/10.1145/3453483.3454032

[54] Dennis Ritchie. 1988. *noalias comments to X3J11*. https://usenetarchives.com/view.php?id=comp.lang.c&mid=PDc3NTNAYWxpY2UuVVVDUD4

[55] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. 14–27. https://doi.org/10.1145/3167092

[56] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 256–270. https://doi.org/10.1145/2837614.2837655

[57] Zachary Tatlock and Sorin Lerner. 2010. Bringing Extensibility to Verified Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. 111–121. https://doi.org/10.1145/1806596.1806611

[58] The Coq Development Team. 2021. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.4501022

[59] The Coq Development Team. 2021. *The Coq Proof Assistant: Reference Manual, version 8.13*. https://doi.org/10.5281/zenodo.4501022

[60] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as Libraries. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 132–141. https://doi.org/10.1145/1993498.1993514

[61] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-Graph Translation Validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 295–305. https://doi.org/10.1145/1993498.1993533

[62] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 17–27. https://doi.org/10.1145/1328438.1328444

[63] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified Validation of Lazy Code Motion. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery,

New York, NY, USA, 316–326. https://doi.org/10.1145/1542476.1542512

[64] Justine Tunney. 2021. *The Byte Order Fiasco*. https://justine.lol/endian.html

[65] Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler Verification Meets Cross-language Linking via Data Abstraction. In *ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 675–690. https://doi.org/10.1145/2660193.2660201

[66] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certifying the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 5, ICFP, Article 84 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473589

[67] Victor Yodaiken. 2021. How ISO C Became Unusable for Operating Systems Development. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems* (Virtual Event, Germany) *(PLOS '21)*. Association for Computing Machinery, New York, NY, USA, 84–90. https://doi.org/10.1145/3477113.3487274

[68] Vadim Zaliva and Matthieu Sozeau. 2019. Reification of shallow-embedded DSLs in Coq with automated verification. In *CoqPL'19: The Fifth International Workshop on Coq for PL*.

[69] Théo Zimmermann and Hugo Herbelin. 2017. Coq's Prolog and application to defining semi-automatic tactics. In *Type Theory Based Tools*. Paris, France. https://hal.archives-ouvertes.fr/hal-01671994