2021

Don't Mind The Formalization Gap: The Design And Usage Of Hs-To-Coq

Antal Spector-Zabusky
University of Pennsylvania

Don’t Mind The Formalization Gap: The Design And Usage Of Hs-To-Coq

Abstract

Using proof assistants to perform formal, mechanical software verification is a powerful technique for producing correct software. However, the verification is time-consuming and limited to software written in the language of the proof assistant. As an approach to mitigating this tradeoff, this dissertation presents hs-to-coq, a tool for translating programs written in the Haskell programming language into the Coq proof assistant, along with its applications and a general methodology for using it to verify programs. By introducing edit files containing programmatic descriptions of code transformations, we provide the ability to flexibly adapt our verification goals to exist anywhere on the spectrum between “increased confidence” and “full functional correctness”.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Computer and Information Science

First Advisor

Stephanie Weirich

Keywords

Coq, Edit files, Haskell, hs-to-coq, Translation, Verification

Subject Categories

Computer Sciences

DON'T MIND THE FORMALIZATION GAP: THE DESIGN AND USAGE OF `hs-to-coq`

Antal Spector-Zabusky

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2021

Supervisor of Dissertation

Stephanie Weirich
Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Commitee

Steve Zdancewic, Professor of Computer and Information Science, Chair
Benjamin Pierce, Professor of Computer and Information Science
Mayur Naik, Professor of Computer and Information Science
Andrew Appel, Professor of Computer Science, Princeton

DON'T MIND THE FORMALIZATION GAP:
THE DESIGN AND USAGE OF `hs-to-coq`

Acknowledgments

First and foremost, I want to extend my deepest thanks to my advisor, Stephanie Weirich. I still remember being an uncertain PhD student coming into her office to change projects and hearing her say, "I have a great project for you". I was nervous it wouldn't be a good fit, but from the moment the next words were out of her mouth I knew it was perfect. She understood me from the get-go, both as a researcher and as a person, and the lessons I took from her mentorship in research, programming, theorem proving, and writing made me the programming language theorist I am today. Stephanie's keen mind, her insightful problem-solving skills, her persistence, her eagerness to get hands on with code, and her patience made her the best advisor I could have asked for. And her passion for board games was crucially important icing on the cake! Without you, Stephanie, this document – and everything it stands for – wouldn't be here.

In a similar vein, I want to extend my thanks to Benjamin Pierce, with whom I worked when I first arrived at Penn. Benjamin knew how to get me involved with research, bringing me into collaborations both inside and outside Penn and ensuring my entry into the programming language research community; he also gave me my first graduate-level lessons in writing good research papers and giving high-quality talks, skills which I have taken to heart and have kept cultivating. Benjamin also helped me through my struggle to find a thesis project that suited me, and was dedicated to helping me find that project no matter who was running it. Thank you, Benjamin, for getting me started, and for knowing when to let me fly.

I also want to thank the rest of my committee, for reading this dissertation and more. Thank you to Steve Zdancewic, who has been there since the beginning; beyond this thesis, he's helped me with many a presentation and sparred with me in many a board game. Thank you to Andrew Appel, whose presence and advice as part of DeepSpec helped bring a different perspective to my research goals. And thank you to Mayur Naik for stepping into the DeepSpec world to understand my work.

Beyond my committee, I have been supported by more people than I can name. Thank you to my Penn PL cohort, who entered the PhD with me: Leonidas Lampropoulos, Jennifer Paykin, and Robert Rand. It was a sincere pleasure to navigate the PhD together with them, from classes to Quizzo to research and finally (finally!) to graduating. I'm glad to call them my colleagues, but I'm even more grateful to call them my friends. And thank you as well to Kenny Foner, Alex Burka, Sonia Roberts, and my other Penn friends for their academic, institutional, and personal support.

In the same vein, my thanks go out to all my other collaborators, because research is not an island. Thank you to the `hs-to-coq` team, without whom the work in this dissertation wouldn't have been possible: Stephanie, Yao Li, Joachim Breitner,

Christine Rizkallah, and John Wiegley. Going back further, thank you to my earliest collaborators, on testing and micro-policies: Benjamin, Leo, Arthur Azevedo de Amorim, Cătălin Hriţcu, John Hughes, Dimitrios Vytiniotis, Nick Giannarakis, and Andrew Tolmach. And of course, my thanks to everyone in Penn PL Club not just for their academic support, but also for making Fridays (and other days) better.

Even before Penn, my thanks to the Williams CS department for nurturing me and my love of computer science. In particular, my thanks to my undergraduate advisor Steve Freund, for teaching me about programming language theory, introducing me to PL research, and for helping me navigate my path forward; and to Jeannie Albrecht, for giving me the opportunity to do my very first computer science research project.

I also want to thank GET-UP for supporting all the graduate students at Penn. I'm proud of what we built in solidarity with each other, and although we didn't win, I believe we can change that next time.

On a less academic note, my thanks to the communities in Philadelphia who supported me throughout this process. Thank you to my Penn Quizzo team, for making sure I was out of the house on Monday nights; to Penn Gamers, for transmuting a shared love of board games into friendships; and to the Thursday night contra dance, for being a broad community that embraced me and kept me moving.

I would not have made it to the finish line without the love and support of all my friends. My wholehearted thanks to the group chat: Colin Killick, Molly Olguín, Mattie Mitchell, Annie Moriondo, Jackie Pineda-Andrews, and Ian Pineda-Andrews (or Agni, Casimir, Ilaina, Max, Screech, and all of Punchworld —Zoltán). Thank you for being there *literally* 24/7. I appreciate their being a neverending fount of camaraderie, serious ideas, silly jokes, and sincere emotional support. My thanks to my Williams undergraduate thesis crew: Matt Hosek, Tori Borish, Katie Kumamoto, and Dan Kohane. I'm glad to have had them along for this journey twice – and we're all finally allowed to sleep now! My thanks to Aaron Bauer, Jake Levinson, and Nick Arnosti, who introduced me to board games. This would be enough of a reason for thanks, but I also appreciate their lasting friendship and its propensity for deep conversation (preferably over a board game or four). And my thanks to Sasha Ehrhardt for being my friend since we were 5 years old, from tigers and gibbons to computers and libraries with a *whole lot more* in between.

My thanks to Caron Bove for driving me between LACS and IHS back in high school so that I could attend my math and science classes – I told you then that I'd acknowledge you now, and I'm delighted to finally be able to do so.

I want to recognize my great-uncle Clifford Spector, whom I wish I'd met. It amazes me that his work in computability theory nestled so closely next to my chosen field without me realizing it, and I'm tickled that he ended up my academic great-great-great-uncle, our familial and professional lineages off by only two generations.

And last, but never least, my neverending thanks and gratitude to my family. Your love, support, and jokes (good and bad) mean more to me than I will ever be able to say. My thanks to my mom and dad, Stacia Zabusky and Donald Spector, for not merely believing in me, but for reifying that belief into actions that build me up. And my thanks to my brother, Elias Spector-Zabusky, for understanding me more deeply than anybody else in the world. I love you all.

ABSTRACT

DON'T MIND THE FORMALIZATION GAP:
THE DESIGN AND USAGE OF `HS-TO-COQ`

Antal Spector-Zabusky
Stephanie Weirich

Using proof assistants to perform formal, mechanical software verification is a powerful technique for producing correct software. However, the verification is time-consuming and limited to software written in the language of the proof assistant. As an approach to mitigating this tradeoff, this dissertation presents `hs-to-coq`, a tool for translating programs written in the Haskell programming language into the Coq proof assistant, along with its applications and a general methodology for using it to verify programs. By introducing *edit files* containing programmatic descriptions of code transformations, we provide the ability to flexibly adapt our verification goals to exist anywhere on the spectrum between "increased confidence" and "full functional correctness".

Contents

List of Figures

CHAPTER 1

Introduction

Writing software is hard. Writing correct software is harder. The history of computing is replete with dire failures and heroic efforts to prevent them. Whether the problem is simple errors in neophytes' first programs, everyday web browsers crashing, or the tragedy of the Therac-25 radiation poisonings (Leveson and Turner, 1993), software bugs are inescapable.

It is a testament to the skill and care of programmers since Ada Lovelace that software, by and large, works. Researchers and practitioners have, over the years, developed a panoply of approaches that we can take to increase our confidence that a program is correct; these approaches include pencil-and-paper reasoning, code review (Baum, Leßmann, and Schneider, 2017), unit testing (Runeson, 2006), property-based random testing (e.g., QuickCheck) (Claessen and Hughes, 2000), model checking (Jhala and Majumdar, 2009), and proof-based formal verification (Barendregt and Geuvers, 2001; Wiedijk, 2006). It is this last that we are concerned with here.

Formal verification, in general, means providing a *mathematical proof* that a piece of software is correct. This can provide a unique level of certainty for the parts of the program covered by the proofs; other techniques can never provide the irrefutability of mathematical proof. At the same time, this means that verification comes with unique challenges, since writing proofs is a particularly difficult task. The form of the proof used in verification varies; it can range from providing a handwritten proof authored by a human who has analyzed the source code, to the model checking approach where the program's behavior on specific families of inputs is exhaustively checked, to the approach that we will discuss here: computer-aided theorem proving. This approach uses a program, called a *proof assistant*, which can both express mathematical theorems and proofs, and also check that a proof of a theorem is correct. The reason these programs are called proof *assistants* is that they do not generate the proofs themselves; instead, the user writes the proof in the language of the proof assistant, perhaps assisted by automation capabilities that it makes available.

Computer science has come a long way, and we now know that verifying code is possible. But *what code* can we verify? This is where research remains in flux. While we can reliably verify code written in proof assistants themselves, this is limiting. In this dissertation, I will present `hs-to-coq`, a tool for translating programs written in the Haskell programming language into the Coq proof assistant so that they can be verified. This translation technique supports verifying *existing* Haskell programs with *realistic* levels of complexity, without ever needing to modify them.

Haskell is a purely functional, statically typed, general purpose programming language. Its type system was originally based on Hindley-Milner type inference over algebraic data types extended with *type classes*, a form of ad hoc polymorphism; it

now supports a far more complex universe of types, including generalized abstract data types, higher-rank polymorphism, and kind polymorphism. Unusually, it is *nonstrict*: terms may be evaluated only when needed, often implemented via lazy evaluation. While not as popular as truly mainstream languages like Java or Python, Haskell programs are written "in the wild", and not simply for academic reasons; the Glasgow Haskell Compiler (GHC) (Marlow and Peyton-Jones, 2012) is one such example, as are the tiling window manager xmonad (Janssen, Stewart, Vogt, Yorgey, Wagner, Roundy, Schoepe, Mertens, Pouillard, Cheplyaka, Branwen, Mai, Shepherdson, and Mullins, 2021) and the document format conversion tool Pandoc (MacFarlane, 2021).

Coq can be viewed in two different ways. First, Coq is a proof assistant: a program designed to allow a user to prove mathematical theorems. Coq can express both theorem statements (written in a functional language called *Gallina*) and mathematical proofs of those theorems (written either in Gallina or in *Ltac*, a "tactic language" for generating proofs); it then checks that these proofs are correct. It is a proof *assistant* because it does not generate proofs (or theorem statements) entirely automatically; it enables the user to write these proofs (and theorem statements), provides abstraction and automation facilities that the user can take advantage of, and ensures the user does not make mistakes. It has been used for a variety of heavy-duty verification projects, such as verifying the four-color theorem (Gonthier, 2008) and writing CompCert, a verified C compiler (Leroy, 2009).

Simultaneously, Coq is a purely functional, statically typed, total programming language with dependent types. Its type system is based on the Calculus of Inductive Constructions, where the "inductive constructions" refer to generalized algebraic data types, and has been extended with further features such as coinduction. Through what is called the *Curry–Howard correspondence*, types can be viewed as theorem statements and terms of those types can be viewed as proofs; the type system of Coq is expressive enough to state meaningful theorems, and because the language is total (that is, because it checks that all programs terminate), the proof system is consistent. Because this proof system is also a programming language, it is particularly well-suited to stating and proving theorems about programs written in Coq (more precisely, in Gallina, but we elide the distinction going forward).

As we can see from these descriptions, at a high level of abstraction, Haskell and Coq are very similar: both are purely functional and statically typed, and both are built on top of generalizations of algebraic data types. But Haskell was designed as a programming language, and Coq was designed as a proof assistant. Is there a way we could get the best of both worlds? If we could link these languages, then we could bring the benefits of formal verification with a proof assistant to Haskell code.

While thinking about how to bridge this gap, we came up with an idea: what if we could take Haskell code and automatically convert it into Coq? This would allow us to verify existing Haskell programs that had not been written with verification in mind, and would allow us to carry out that verification with all the power of the existing Coq tooling. To evaluate this approach, I wrote the first version of `hs-to-coq`, a tool for doing just this: translating Haskell programs into Coq. The hope was that we could then verify the translated Haskell program just as we could verify any other Coq program. And once the translation tool was complete, this was borne out: we

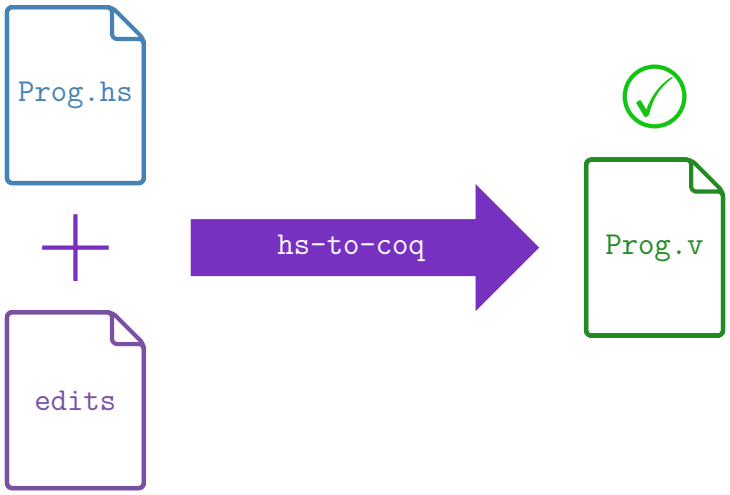

FIGURE 1.1. How `hs-to-coq` operates: Given a Haskell file (`Prog.hs`), the user writes edit files (`edits`) which record the changes that will be made to the generated output. Once they have both of these, the user feeds them into `hs-to-coq`, examines the generated Coq output (`Prog.v`), and proves it correct.

were able to state and prove theorems about simple Haskell programs just as we had hoped!

The next question was how to scale this process up. Due to the intricacies of the translation process, `hs-to-coq` was already nontrivial, and as we scaled it up to support more complex Haskell programs, we discovered commensurately more complexity and richness within the translation task, leading to a novel perspective on verification. These more complex programs correspondingly had many more ways we could verify them: different pieces to translate, different methods of translation, and different theorems to prove. To support this variety of different approaches, and to more broadly be able to render the Haskell code into a form that Coq could work with, we gave `hs-to-coq` the ability to transform the Haskell program during the translation process in a user-configurable way.

With this, we come to the full picture of how to use `hs-to-coq`, which is also presented diagrammatically in Figure 1.1: after selecting the Haskell code they wish to verify, the user also writes a new *edit file* containing a programmatic representation of the systematic code transformations that they wish to apply to the output of the translation. Once they have the Haskell code and the edits, they pass these both to `hs-to-coq`, which produces the appropriately modified Coq file, on which verification can then proceed. During development, attempting the verification process often results in the user realizing that they need further edits, producing an edit–verification feedback loop.

These edit files are a key contribution of `hs-to-coq`. They provide support for `hs-to-coq`'s unique approach to verification, which has two major prongs:

(1) Enabling the user to *never edit Haskell code*, instead using edit files to make necessary changes.
(2) Providing a *mechanized formalization gap*.

The first prong is `hs-to-coq`'s response to the phenomenon that, in practice, any attempt to verify existing code will need to modify said code to make it suitable for verification. These modifications can range from changing variable names to avoid a reserved word in the verifier (trivial) to making sure that a function is amenable to the termination checker (difficult). In practice, making these sorts of modifications to some degree is always necessary. Edit files record these modifications in a declarative format. This is essential because making these sorts of changes directly to the affected source code isn't practical. For one thing, it makes it very difficult to understand what has changed from the original. Text-level diffs are helpful, but only tell us so much; as a particularly striking example, changing the name of a variable can affect large swaths of the program even though the change is conceptually simple. Moreover, it also becomes impractical to pull in updates made to the original source code, so the verification rapidly becomes targeted at a fork of the original project. Editing the output of the translation is even worse; it has all these disadvantages, but also prevents ever running the translation a second time, as that would require redoing all the changes by hand.

By using edit files, we never need to edit Haskell code; any change we would make is recorded as a single instruction in such a file. Verification remains targeted at the exact code we started with. We also can understand the changes that are being made more directly: each edit corresponds to a single semantic change. And we also ensure that this is an accurate and complete list of all the changes being made, because we never edit code in any other way.

The second prong is a different way of looking at these same edit files. When we verify a model of any program – and our Coq translations are in a certain sense a model of the original Haskell programs – there is always some *formalization gap* between the theorem and the actual running code. In addition to trusting the pieces of the system (our theorem prover, the hardware, etc.), we have to trust that our model corresponds to the original program. Usually, this analogy is discussed in prose, explaining why we believe the two are connected; if a mechanical translation is used, then any changes that were made directly to the input to enable translation are invisible. For `hs-to-coq`, however, the edit files serve as a record of exactly what changes were made to the input: which functions were skipped, rewritten, and so on. This record is also *machine-readable*, since `hs-to-coq` uses it as input to perform the translations. Thus, the formalization gap between our Coq model and Haskell input has now been mechanized; any changes are recorded in the edit file itself.

1.1. How to work with `hs-to-coq`

Now that we have outlined `hs-to-coq`'s approach to verifying code, we can walk through the experience step by step. The workflow, which was summarized in Figure 1.1, has four steps:

(1) Find (or write) some Haskell code.
(2) Write edit files describing the modifications being made to the generated code.
(3) Run `hs-to-coq` to generate Coq output.
(4) Prove theorems about the resulting Coq library.

By walking through what each of these steps entails, we'll gain a fuller perspective on what using `hs-to-coq` is like and why we might choose to do so. As our guide, we'll use a the `merge` function from mergesort. This function merges two sorted lists into a single sorted list; for example, we will have `merge [1,3,5] [0,2,4] == [0,1,2,3,4,5]` and `merge [0,4] [1,2,8] == [0,1,2,4,8]`.

Step 1: Find Haskell code. In order to verify a program, the first thing we have to do is choose what we're verifying. In most of our actual work, this has involved choosing existing Haskell programs; to keep things manageable for this example, we'll write the `merge` function right now.

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y    = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

We saw some examples of this function's behavior above; recall that we're only interested in what happens if the input lists are sorted. We'll assume that this function lives in a module named `Sort`.

Step 2: Write edit files. Unfortunately, we can't simply translate `merge` into Coq automatically. If we try, we get the following Coq output:[1]

```
Definition merge {a} `{Ord a} : list a -> list a -> list a :=
  fix merge (arg_0__ arg_1__ : list a) : list a :=
    match arg_0__, arg_1__ with
    | nil, ys => ys
    | xs, nil => xs
    | cons x xs, cons y ys =>
      if x <= y : bool
      then cons x (merge xs (cons y ys))
      else cons y (merge (cons x xs) ys)
    end.
```

We can see that this Coq function lines up directly with the Haskell function: the name `merge` remains `merge`; the Haskell type `[a]` becomes the corresponding Coq type `list a`; and the pattern matches on `[]` and `x:xs` or `y:ys` become a pattern matches on `nil` and `cons x xs` or `cons y ys`, which is how Coq spells its list constructors. At

[1] Here and throughout this dissertation, we will often simplify the output by adjusting whitespace and removing module prefixes.

the same time, `hs-to-coq` has had to make some accommodations for the difference between Haskell and Coq: rather than sometimes using `Definition` and sometimes using `Fixpoint` (Coq's top-level keyword for defining recursive functions), `hs-to-coq` always uses `Definition` and puts `fix` (Coq's syntactic construct for recursive functions) at the term level; `hs-to-coq` also has to synthesize the variable names `arg_0__` and `arg_1__` and then use a `match` expression, since Coq does not have Haskell's equational-style pattern matching.

Trying to compile this function, however, produces the following Coq error:

> Error: Cannot guess decreasing argument of fix.

This means that Coq cannot tell that this function is guaranteed to terminate, and we can see why by examining the recursive calls to `merge`. Coq requires that every fixpoint be defined by recursion on a single argument; that is, every recursive call must be to a subterm of that argument. However, `merge` alternates between recursing on its first argument, in the call `merge xs (cons y ys)`, and its second, in the call `merge (cons x xs) ys`. This recursion pattern clearly terminates, since exactly one list shrinks at each step, but Coq cannot see that this is so.

This is exactly the sort of situation that requires edit files. For real code, these edit files can get profoundly complex. In this particular case, our edit file must instruct `hs-to-coq` to make changes to the code so that Coq can tell it terminates. That looks like the following:

```
termination Sort.merge {measure (length arg_0__ + length arg_1__)}
obligations Sort.merge Tactics.program_simpl; simpl; lia
```

Here, we see our first introduction to edits. Informally, these two edits, taken together, say "you can prove that `merge` is terminating by showing that the sum of the lengths of the inputs decreases on every recursive call, and here's how."

Edits begin with the name of the edit, which can be one or more words (e.g., `termination`, `obligations`), followed by some arguments appropriate for the edit in question. Here, we need both the name of the definition to alter, and then either the termination measure (for `termination`) or the proof script (for `obligations`). The `termination` edit alters the translation to use Coq's `Program` commands (Sozeau, 2006), which themselves can automatically elaborate a Coq term to provide a more interesting termination argument. The `measure` keyword in the edit corresponds to the same keyword in Coq, and specifies the value that will decrease on every recursive call. Note that it is specified in terms of the translated positional argument names.

For `obligations`, after we specify the name of the function, we provide the Coq tactics that will be used to prove any proof obligations that `Program` introduces for us; the tactics used here will prove that the sum of the lengths of the input lists really does decrease on every recursive step. The first two tactics perform two different kinds of simplification; the `lia` tactic is a solver for linear integer arithmetic.

Step 3: Run `hs-to-coq`. Now that we have our edits, we can run `hs-to-coq` on *both* our Haskell code *and* our edits together. This produces the following Coq output, which compiles successfully:

```
Program Fixpoint merge
    {a} `{Ord a} (arg_0__ arg_1__ : list a)
    {measure (length arg_0__ + length arg_1__)} : list a :=
  match arg_0__, arg_1__ with
  | nil, ys => ys
  | xs, nil => xs
  | cons x xs, cons y ys =>
    if Bool.Sumbool.sumbool_of_bool (x <= y)
    then cons x (merge xs (cons y ys))
    else cons y (merge (cons x xs) ys)
  end.

Solve Obligations with (Tactics.program_simpl; simpl; lia).
```

If we compare this with our output in step 2, we'll see that things are very similar; the differences are that (1) we begin with `Program`; (2) a termination `measure` has been inserted, which we saw in the `termination` edit; (3) we end with `Solve Obligations`, using the tactics from the `obligations` edit; and (4) the `if` expression uses the `sumbool_of_bool` function on the condition. Differences 1–3 are the ones we discussed above, when talking about the edit file; the use of `sumbool_of_bool` also comes from these edits, and is inserted whenever the output uses `Program`. It allows the proof obligations in each branch to learn about which value the conditional took, which – though not important here – is sometimes critical.

Step 4: Prove theorems. Finally, we can actually attempt to verify `merge`, in the same way that we would verify any other Coq program. For real code, this step is challenging, as verification always is. Here, then, we will present the statement of the theorem that if the two input lists to `merge` were sorted, then the output list will also be sorted:

```
Theorem merge_preserves_sorted `{OrdLaws a} (xs ys : list a) :
  Sorted xs ->
  Sorted ys ->
  Sorted (merge xs ys).
Proof.
  induction xs as [|x xs IHxs];
  induction ys as [|y ys IHys].
  ...
Qed.
```

This theorem statement uses the type class `OrdLaws`, which we provide along with `hs-to-coq`; if a type `T` has an instance of `OrdLaws`, this means that `T` has an instance of the Haskell `Ord` class, and that this instance is actually a total order. For more information, see Section 5.1.

1.2. The edit language and the mechanized formalization gap

We've now seen how to take `hs-to-coq` through its paces, from start to finish. We've seen how `hs-to-coq` allows us to take existing (or bespoke) Haskell code, customize the translation as necessary, and then verify it in Coq. One thing that may have become apparent is that step 2, writing the edit files, is a deeply involved process. Moreover, it's a process of a different nature: while verifying Coq code, as required in step 4, is difficult, those of us in this field are used to it. Writing edit files is a challenge unique to `hs-to-coq`, but it is through edit files that we obtain the full power of our mechanized formalization gap. The specific benefits of edit files are manifold:

(1) We can verify existing Haskell code.
(2) We never need to edit Haskell code by hand.
(3) We get a mechanized formalization gap.
(4) We can attain confidence instead of perfection.

These last two items are the most important, and perhaps require more explanation. The notion of a *mechanized formalization gap*, mentioned previously, is a key contribution of `hs-to-coq` and its development model. Rather than constantly strive to minimize the difference between the original Haskell input and the verified artifact, we embrace the formalization gap between them, and we allow it to be widened or shrunk as necessary. This makes the process of verification much smoother, as we can control exactly how much detail we arrive at. But this is only sustainable because our formalization gap is *mechanized*, because it is recorded entirely in the edit files. These files are machine-readable (hence "mechanized"), and, in combination with the transformation process of `hs-to-coq` itself, reflect the precise difference between the original Haskell and the verified Coq. These files thus *are*, in a certain sense, the formalization gap: they are a complete record of all the differences between the original implementation and the verified code. And because we have this complete record, we can more confidently allow users to make the variety of changes that they do.

This dovetails nicely with the last benefit, that of aiming for confidence instead of perfection. Because `hs-to-coq` decouples proof from translation, there is no requirement to prove everything about a single piece of code. At the same time, its flexible edits allow for very radical transformations of code, perhaps removing obstacles entirely. This gives us a choice: by choosing our edits and our theorems carefully, we could prove complete functional correctness of unmodified Haskell code; alternatively, we could simply translate Haskell code as best we could, modify it with edits, and prove no theorems but at least gain the confidence that our translated code terminated. Anywhere on the spectrum from "full formal verification" to "using Coq for greater confidence" can be valuable, and `hs-to-coq` does not commit us to a single point.

For these reasons, edit files provide a unique perspective on verification, and one which is not bound to `hs-to-coq`. In this dissertation, we explore how `hs-to-coq` and its edit language enable verifying Haskell by using Coq; however, the broader perspective of incorporating a mechanized formalization gap into verification approaches

is not bound to the specifics of this tool, and it is my hope that the details of our experience are generalizable to the broader domain of software verification.

1.3. DeepSpec

This work is part of the DeepSpec project (Appel, Beringer, Chlipala, Pierce, Shao, Weirich, and Zdancewic, 2017), which is based on developing "the science of deep specification". This is a broad research effort focused on the question of how to write correct software, and on an answer to that question that revolves around a notion of a *deep specification*: a specification that is *rich*, *two-sided*, *formal*, and *live*. The DeepSpec project defines these terms as follows:

> - *rich* (describing complex component behaviors in detail),
> - *two-sided* (connected to both implementations and clients),
> - *formal* (written in a mathematical notation with clear semantics to support tools such as type checkers, analysis and testing tools, automated or machine-assisted provers, and advanced IDEs (integrated development environments)) and
> - *live* (connected via machine-checkable proofs to the implementation and client code).
>
> —Appel et al. (2017); bullets added

We want to ensure that the specifications we write for Haskell code are themselves *deep* in this sense. Our work in particular tries to ensure that *formal* and *live* come automatically:

Formal: This comes with our decision to work in Coq. As all of our specifications and proofs are in Coq, they are formal by definition.

Live: Because `hs-to-coq` translates existing Haskell implementations to Coq, the proofs we write about the Coq programs are mechanically connected to the original code. Coq connects the proof to the Coq program, and `hs-to-coq` and the edit files connect the Coq program to the Haskell original.

One difference in the notion of "live" that comes from `hs-to-coq` and the original definition of "live" is that we *don't* have machine-checkable *proofs* all the way from the Haskell implementation to the Coq proofs. Because `hs-to-coq` is unverified, and because edit files allow for significant reinterpretation of the Haskell code, we merely have a "strong resemblance". However, we argue that this is still a valuable form of being live. While it would be wonderful to have a fully machine-checked proof all the way through, other forms of guarantees still provide confidence. Because the connection to the original Haskell is fully automatic, we can reapply our proofs to newer versions of the Haskell code. This process won't be *free* – we still have to update our proofs when the code changes. But we never need to worry that our Coq implementation and the Haskell implementation have drifted out of sync. If they are out of sync, it is because our edits deliberately made them so.

Furthermore, if a user wants the extra assurance that they are only working with mechanically verified code, there is always another option: using the Coq implementation we produced. This Coq implementation can be used in multiple different ways:

- it can be evaluated with Coq's built-in evaluators;

- it can be extracted to OCaml, Haskell, or Scheme through Coq's built-in extraction mechanism; or
- it could be compiled with Certicoq (Anand, Appel, Morrisett, Paraskevopoulou, Pollack, Bélanger, Sozeau, and Weaver, 2017) once Certicoq is able to handle all the features of Coq that appear in the output of `hs-to-coq`.

Each of these techniques removes the Haskell implementation from consideration and removes `hs-to-coq` from the trusted computing base; instead, one of these other, leaner tools can be trusted instead.

1.4. Contributions

Software verification is a vast and well-explored landscape. The particular contributions of this thesis are as follows:

- I, along with my collaborators, built the `hs-to-coq` tool for translating Haskell into semantically analogous Coq, allowing for Coq-based verification of *existing* Haskell code that was not written to be verified.
- We introduced *edit files*, a textual language describing changes that `hs-to-coq` will make to the generated code, and present a methodology that uses these so that we never edit Haskell code.
- We embraced having a *mechanized formalization gap*, with this embrace including both the "mechanized" and "formalization gap" parts. Both of these pieces come from embracing edit files. The former comes about because edit files allow the formalization gap between the Haskell input and verified Coq output to be precisely specified in a machine-readable way. The latter comes about because edit files allow for a fluidity and flexibility in choosing exactly what the code that we target for verification will look like.
- We use the flexibility offered by edit files to embrace a verification ethos of attaining confidence, not perfection. Every line of Coq we write removes places for bugs to hide in the Haskell, even if we cannot guarantee complete functional correctness.
- We applied the above methodology to verify significant portions of the `base` and `containers` library, extending `hs-to-coq` along the way.
- We have begun verification of GHC itself, targeting its internal language Core and some Core-to-Core transformations.

This work is not without its limitations. To begin with, Haskell is partial, and Coq is not. We provide a variety of techniques to attempt to blur this distinction, but ultimately must change the semantics of the input program to account for this. In a similar vein, Haskell, being lazy, only defines coinductive types and uses corecursion; unless instructed otherwise, `hs-to-coq` treats all types as inductive and all self-reference as recursion. We believe that this more often captures programmer intent, but it does not model Haskell precisely. We also lack a formal semantics of Haskell, and thus cannot make our claim of semantic correspondence between the two precise; even if we had these, such a claim would be a Herculean task and beyond the scope of this work. Lastly, we use the flexibility offered by edit files to embrace a verification ethos of attaining confidence, not perfection. This is repeated from above because it is

both a contribution and a limitation. Not striving for complete functional correctness means that this strongest form of Coq verification is beyond us.

Finally, this work raises questions that need to be addressed. First, why should a user trust `hs-to-coq`'s translation, if we cannot provide a proof of `hs-to-coq`'s correctness? One of our answers to this is *legibility*: a focus on making sure that the input Haskell and output Coq are visibly very similar. We discuss this further in Section 3.2.1. Second, what does it mean to have a Coq proof of something? We know exactly what formal Coq proposition the proof is of, but how does that correspond to our intuitive goals? This is a perennial challenge of any formalization effort, and `hs-to-coq` is no different. Our best approach is to leverage DeepSpec's notions of deep – rich, two-sided, formal, and live – specifications, as we saw in Section 1.3.

1.5. Outline

In this dissertation, I explore how I (and my collaborators) designed, built, and applied `hs-to-coq`.

- In Chapter 2, I present an expanded example of using `hs-to-coq`.
- In Chapter 3, I present the broader design and usage of `hs-to-coq`, helping explain both our goals for it and some of the technical details of how it functions.
- In Chapter 4, I present the design of the *edit language* that makes up the mechanized formalization gap.
- In Chapters 5–7, I discuss how we built up `hs-to-coq` and applied it in practice. Through the lens of multiple case studies, I demonstrate the breadth of applications of `hs-to-coq`, building up to our present focus on the power of flexible mechanized formalization gaps. These chapters discuss results from our three papers, "Total Haskell is reasonable Coq" (Spector-Zabusky, Breitner, Rizkallah, and Weirich, 2018), "Ready, `Set`, verify! Applying `hs-to-coq` to real-world Haskell code (experience report)" (Breitner, Spector-Zabusky, Li, Rizkallah, Wiegley, and Weirich, 2018), and "Embracing a mechanized formalization gap: Pragmatic software system verification (extended version)" (Spector-Zabusky, Breitner, Li, and Weirich, 2019, unpublished), respectively.
- In Chapter 8, I discuss in detail each of the 34 distinct edits that `hs-to-coq` supports.
- In Chapter 9, I discuss some of the other work in the field that connects to `hs-to-coq`.
- And finally, in Chapter 10, I wrap up what we've seen, evaluate the edit language, and look to potential avenues to extend and apply `hs-to-coq` in the future.

CHAPTER 2

An Introductory Example: Bags

Using `hs-to-coq` involves working with a number of moving parts: Haskell code, `hs-to-coq` edit files, generated Coq code, Coq specifications, Coq theorems, and Coq proofs. We saw how these parts fit together in the introduction, but only in the context of the single `merge` function. In order to understand `hs-to-coq` better, we'll now work through a more complete example: the verification of a simple data structure. In this chapter, I will walk through the original Haskell code, how we translate it, the resulting Coq, the theorems we state, and the proofs we write. Much work in functional programming centers around data structures and their invariants, so this work is a good example of the sorts of techniques that are necessary for more complicated structures.

The particular data structure I will present here is nominally a *bag*, or unordered multiset. The bags $\wr 1, 2, 2, 3, 3, 3 \int$ and $\wr 3, 3, 3, 2, 2, 1 \int$ are the same, but the bag $\wr 1, 2, 3 \int$ is different. The particular implementation of bags in question is from GHC itself, and was verified as part of our work on "Total Haskell is reasonable Coq" (Spector-Zabusky et al., 2018); I present the results of this verification in more detail in Section 5.3. However, this implementation of "bags" turns out to be overdetermined: GHC exposes functions that provide an ordering on the bags, meaning that the data structure implemented by its `Bag` type is really a sequence, like a list but with different asymptotics.

Once we have worked through the definition of bags (sequences) in GHC (Section 2.1), we will see how we need to go about translating it: what happens automatically (Section 2.2), and what we need to customize with edits (Section 2.3). We will then look at how to provide a specification for bags and the operations on them, and look at some of the specifications for individual operations (Section 2.4). Finally, having built up the whole process of using `hs-to-coq` in this chapter, we close by reviewing that process so we can see it all in one places (Section 2.5).

This chapter uses the verification of `Bag`s as an example to present the usage of `hs-to-coq`; this is fitting, since it was one of our first verification projects. However, there is more to say about this verification project and the results we learned from it; for more details on this, see Section 5.3, where they are presented in context.

2.1. Bags in GHC

The data structure that GHC defines to implement a bag is a tree of elements and lists, and is presented in Figure 2.1. The data type comes with two invariants, both around emptiness: first, that no recursive occurrence of a `Bag` may be empty; and second, that if we are embedding a list, we cannot embed the empty list. This means that we know two things: first, the only empty `Bag` is `EmptyBag`; and second, nonempty

```
data Bag a
  = EmptyBag
  | UnitBag a
  | TwoBags (Bag a) (Bag a) -- INVARIANT: neither branch is empty
  | ListBag [a]             -- INVARIANT: the list is non-empty
    deriving Typeable
```

FIGURE 2.1. The `Bag` data type in GHC, which implements either a bag (an unordered multiset) or a sequence, depending on your perspective.

`Bag`s are nonempty binary trees with elements (or nonempty lists of elements) at the leaves.

This representation is simple; at the same time, verifying it requires solving three different challenges that are very common in data structure verification:

(1) It has *multiple internal structures* that correspond to the same user-facing value.
(2) It has *internal invariants*, as specified in the comments.
(3) It has a *specification* that must be defined both informally and formally.

This makes it particularly suitable as an introductory example to verifying code while using `hs-to-coq`.

In particular, let us look at challenge (1), the multiple internal structures this data type has. There are many different ways we could change the structure of a `Bag Int` without affecting the collection of values it contains: we could change the associativity of the tree, we could introduce or remove the `ListBag` constructor, or we could combine the `ListBag` and `TwoBags` constructors. For example, five different possible representations of the bag $\{\!|1, 2, 3|\!\}$ are shown in Figure 2.2.

One operation we left out, however, is changing the order of elements in the bag: what about `TwoBags (TwoBags (UnitBag 3) (UnitBag 2)) (UnitBag 1)`? After all, we discussed the mathematical definition of a bag above, and a comment at the top of the module defines "Bag" similarly:

> Bag: an unordered collection with duplicates
>
> —GHC, the `Bag` module, line 6

Thus, it would seem like we could consider that value equivalent. However, as mentioned above, a more careful look at the module reveals that this isn't true. No functions on `Bag`s require an `Ord` constraint or anything similar, yet GHC exposes several functions that depend on or reveal the order of elements in a `Bag`. In particular, the `bagToList` function converts the internal tree representation into a list via a left-to-right (in-order, depth-first) traversal. Lest we think that this is an accident or otherwise not to be relied on, the `Bag` module provides separate `consBag` and `snocBag` functions for, respectively, prepending and appending a single element to a bag; indeed, when we examine the module closely, we see that this left-to-right order is preserved by all the functions that operate on `Bag`s. This means that, instead of thinking about `Bag`s as modeling mathematical bags, we will think of them as modeling *lists*; the

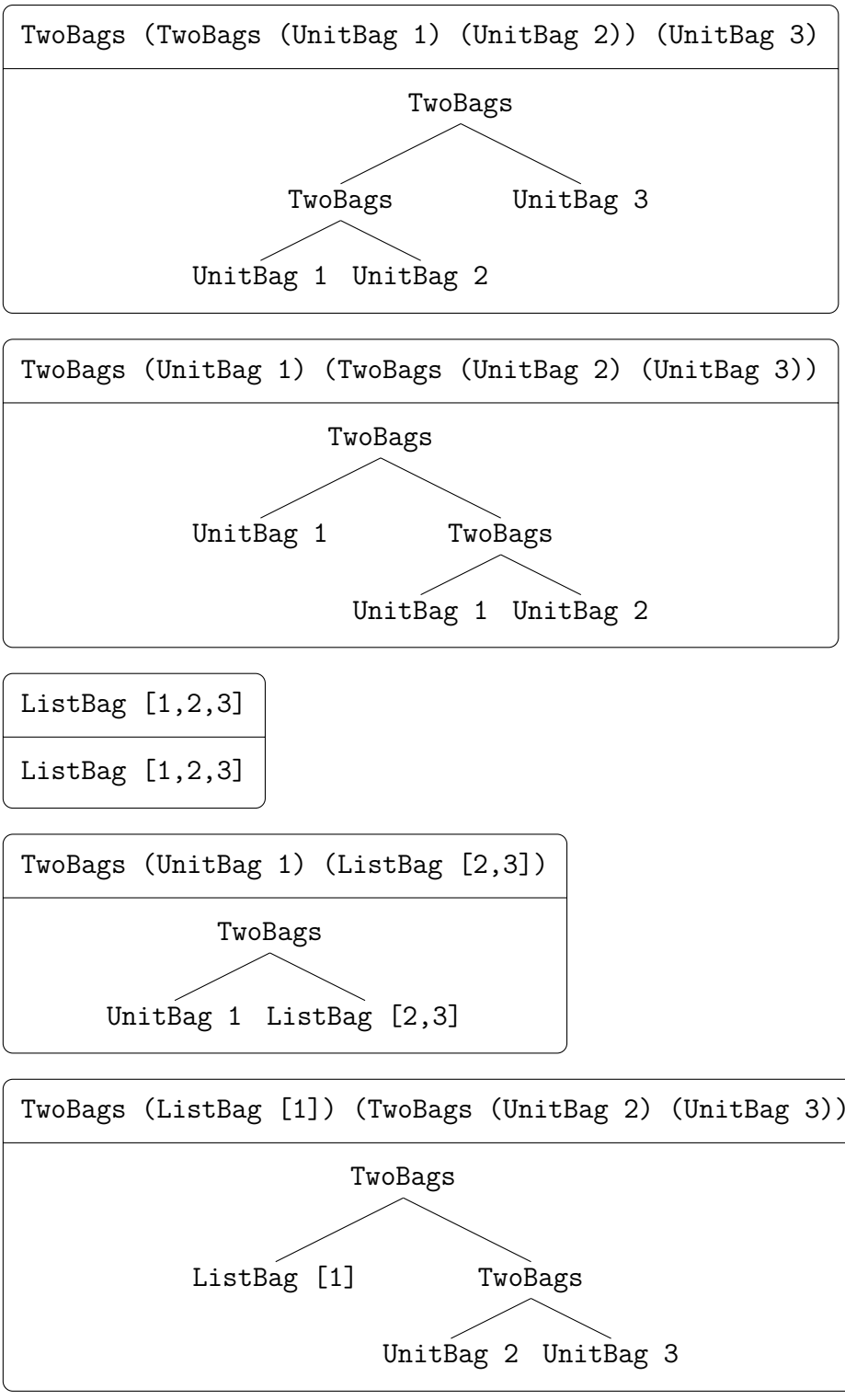

FIGURE 2.2. Five different `Bag` representations of $\{\!|1, 2, 3|\!\}$. Each box contains one representation, with the Haskell term above and the tree structure below.

```
Inductive Bag a : Type
  := EmptyBag : Bag a
  |  UnitBag : a -> Bag a
  |  TwoBags : (Bag a) -> (Bag a) -> Bag a
  |  ListBag : list a -> Bag a.

Arguments EmptyBag {_}.
Arguments UnitBag {_} _.
Arguments TwoBags {_} _ _.
Arguments ListBag {_} _.
```

FIGURE 2.3. The `Bag` data type from GHC translated into Coq by `hs-to-coq` (with some minor reformatting).

above `Bag`s are in fact representations of `[1,2,3]`. The difficulty of determining this denotation is an instance of the general difficulties faced in solving challenge (3), the design of specifications for a type.

2.2. Translating `Bag` and its operations

The `Bag` data type itself can be translated to Coq without any extra configuration, and looks much the same; it is presented in Figure 2.3. The names and types of the constructors are unchanged, and `hs-to-coq` automatically translates Haskell's list type `[]` to Coq's default list type `list`.

The Haskell file, naturally, defines several operations on `Bag`s: creating empty bags, creating singleton bags, appending two bags, and so on and so forth. All of these operations are so straightforward, and have such simple definitions, that we need to make almost no changes to them; they will be translated successfully with the only change being in the representation of numbers. For example, consider the `unionBag` function, which takes the union of two `Bag`s; recalling that `Bag`s are really ordered, we can instead say that this appends two bags.

```
unionBags :: Bag a -> Bag a -> Bag a
unionBags EmptyBag b = b
unionBags b EmptyBag = b
unionBags b1 b2      = TwoBags b1 b2
```

In order to preserve the invariant that the empty bag is never a subterm of another bag, we must pattern-match on the arguments. This translates to Coq directly:

```
Definition unionBags {a} : Bag a -> Bag a -> Bag a :=
  fun arg_0__ arg_1__ =>
    match arg_0__, arg_1__ with
    | EmptyBag, b => b
    | b, EmptyBag => b
    | b1, b2 => TwoBags b1 b2
```

```
end.
```

The difference between Haskell's equational style of pattern-matching and Coq's pattern-matching `match ... with ...` expression aside, we can see that the three pattern matches are reflected exactly in the definition of `unionBags` in Coq.

2.3. Edits for `Bags`

The definitions of the functions that operate on `Bags` are all so simple that, for the most part, manual intervention via edit files is unnecessary. However, nothing is quite *that* simple. The most important edits affecting the `Bag` module are those that come from its surroundings. `Bag` is part of GHC, and GHC is much bigger than just this one data structure. When working on GHC, we thus use two sources of edits: *global* edits, which are used when translating every source file; and *local* or *per-module* edits, which are used when translating specific modules. The global edits for GHC are mostly about two things: skipping things we don't want to translate; and renaming values, either for simplicity or for name collisions.

For `Bag`, the relevant global edits are those that skip code. Principally, `Bag` imports the module `Outputable`, which is responsible for pretty-printing. This is not code that we ever plan to verify – we are not interested in the text of the error messages that GHC produces. Consequently, in every module, we include the edit

```
skip module Outputable
```

This edit suppresses the import of the module, but references to it may remain: there may be instances of the eponymous class `Outputable` that is defined therein. This class is for types that can be pretty-printed; since we don't care about output in general, we don't care about these instances in particular, and so we we suppress all of them with the edit

```
skip class Outputable.Outputable
```

The `skip class` edit suppresses both the named type class and all instances of it, but it does not suppress other references to it or its methods. While modules other than `Bag` rely on further `Outputable`-suppressing edits to handle this, `Bag` only defines an instance of `Outputable` for `Bag`, so these two edits are enough. (`Bag` also imports other GHC-local modules, but they are translated separately and, much like `base`, have their own edits – or not – as necessary.)

There are only two local edits required to translate the `Bag` module itself:[2]

```
rename type GHC.Num.Int = nat

in Bag.lengthBag ↩
  rewrite forall xs, ↩
    Data.Foldable.length xs = ↩
    BinInt.Z.to_nat (Data.Foldable.length xs)
```

[2]Edits are written on one line, but we often include line breaks due to page width constraints; the "↩" marker signifies where we have done so.

The first edit changes all reference to Haskell's fixed-width signed `Int`s to Coq's unbounded nonnegative `nat`s. This change is made in many places in the GHC code base; when `Int`s correspond simply to counting, we do not concern ourselves with overflow. This is a deliberate verification choice, and a portion of the formalization gap that could be closed later with extra effort. We choose to let it stand, as the question of overflow is of low importance in many cases; for instance, the only place `Int` – now `nat` – is used in `Bag` is for the output of `lengthBag :: Bag a -> Int`. We will never be storing a 2^{64}-element `Bag` in memory, so overflow will never be an issue here.

However, the second edit is there to fix this choice of representation coming back to bite us. Haskell programmers use `Int` for multiple different things: sometimes as an efficient representation of integers, sometimes as a type of unique identifiers , sometimes to represent data type sizes (where they will only be nonnegative, hence our choice of `nat`), and so on. In Coq, however, we are more likely to want different types for these different purposes, and the user can use `rename type` to control the selection. In `Data.Foldable`, we chose to represent Haskell `Int`s as Coq `Z`s. The type `Z` is Coq's default type for unbounded signed integers; we chose it because it seemed to be the most useful general purpose default. This choice matters here because the definition of `lengthBag` uses `Data.Foldable.length :: Foldable t => t a -> Int`. This means that `lengthBag`, which is in `Bag`, returns a `nat`, but its implementation uses a `Z`. Thus, this second edit, which rewrites the code in a limited scope:

- `in Bag.lengthBag` scopes the following edit to just apply within the definition of `Bag.lengthBag`;
- `rewrite` is an edit that replaces one translated Coq expression with another wherever the former appears;
- `forall xs,` says that `xs` is a metasyntactic variable for this rewrite; and
- `Data.Foldable.length xs = ↩`
  `BinInt.Z.to_nat (Data.Foldable.length xs)` is the rewriting equation, which wraps calls to `length` with the conversion function from `Z` to `nat`.

As we see here, one downside of our flexibility in choosing representations is that different pieces of the code base use different representations, and then have to interoperate.

2.3.1. Termination. The lack of need for extensive edits makes `Bag` simple to work with. At the same time, however, it is unrepresentative of our usual verification experience. Many files we verify with `hs-to-coq` are not so cooperative, and require some manual intervention before Coq will accept them, often (but not only!) due to questions of termination – in our experience, although most functions are indeed structurally recursive, every large project needs some `termination` edits. While `Bag` is full of recursive functions, the natural way to write all of them was structurally recursive, and so translation was unusually straightforward. This was my initial hope when designing and developing `hs-to-coq`: that idiomatic Haskell code would often look like idiomatic Coq code. And while the Haskell code out there in the wild has needed us to extend `hs-to-coq` far beyond the need for strict structural recursion,

modules like `Bag` are a reminder that, in fact, there is plenty of Haskell code out there that fits into Coq nicely and neatly.

2.4. Specifying the behavior of `Bags`

Now that we have Coq implementations of our various `Bag` operations, we must prove that they are correct. But before we can prove that anything is correct, we need to know what that *means*. For `Bags`, we elect to provide them with a specification in terms of a denotational semantics. Recall from Section 2.1 that these bags effectively model *lists*, because the functions that GHC provides can distinguish between `Bags` whose elements occur in different orders. Conveniently for us, GHC provides the function

```
bagToList :: Bag a -> [a]
bagToList b = foldrBag (:) [] b
```

which converts a `Bag` into a list via a left-to-right (in-order, depth-first) traversal; we take this as our denotation function. (Note that the `foldrBag` function is one of the aforementioned functions that exposes the order of elements in a `Bag`.)

Not all elements of the `Bag` type are well-formed, however; Section 2.1 documented the invariants that `Bags` are required to maintain, which were specified in the comments attached to the definition of the data structure. Specifically, the invariants limit the appearance of emptiness: they require that `TwoBag` never contain an `EmptyBag` and `ListBag` never contain `[]`. This means that the only empty bag is `EmptyBag`, and that `EmptyBag` never appears deeply within a `Bag`. We can encode this invariant as a Coq predicate:

```
Fixpoint well_formed_bag {A} (b : Bag A) : bool :=
  match b with
  | Mk_EmptyBag              => true
  | Mk_UnitBag _             => true
  | Mk_TwoBags Mk_EmptyBag _ => false
  | Mk_TwoBags _ Mk_EmptyBag => false
  | Mk_TwoBags l r           => well_formed_bag l &&
                                well_formed_bag r
  | Mk_ListBag []            => false
  | Mk_ListBag (_ :: _)      => true
  end.
```

The invariant on bags is sufficiently simple that we can encode it as a function into `bool`, but more complex invariants may require the use of (or just be more simply stated in terms of) an inductive proposition.

Thus, verifying `Bags` requires verifying not just correctness, but also well-formedness. Not only do we need to verify that each operation on `Bags` corresponds to the equivalent operation on lists (correctness), but we also need to verify that each operation that produces `Bags` produces well-formed `Bags` (well-formedness). In both cases, we may require that any input `Bags` are well-formed, but this will not always be necessary.

As an example, recall the definition of `unionBags` presented in Section 2.2. The first theorem we need to prove is that, given well-formed `Bags` as input, `unionBags` produces a well-formed `Bag`. The Coq theorem is

```
Theorem unionBags_wf {A} (b1 b2 : Bag A) :
  well_formed_bag b1 -> well_formed_bag b2 ->
  well_formed_bag (unionBags b1 b2).
Proof. case: b1; case: b2 => * //=; intuition. Qed.
```

with the proof follows straightforwardly from case analysis.

The second theorem we prove is that the denotation of `unionBags b1 b2` is the same as the corresponding list operation. What is the list operation that corresponds to union? Since `Bag` is actually the type of lists, this operation is simply list concatenation, `(++)`, and so the theorem in Coq is

```
Theorem unionBags_ok {A} (b1 b2 : Bag A) :
  bagToList (unionBags b1 b2) = bagToList b1 ++ bagToList b2.
Proof.
  by case: b1 => *; case: b2 => *; rewrite -bagToList_TwoBags.
Qed.
```

This can also be summarized as saying that `bagToList` is a homomorphism from bags to lists, although this is an informal manner of speaking.[3]

2.5. From program to theorem

We have now seen in some detail the life cycle of a simple translation and verification effort using `hs-to-coq`. We close out this example by summarizing what this process looks like all in one place, so that we can get a good picture of the flow of using `hs-to-coq`.

(1) First, we start with Haskell data structures and functions, such as

```
data Bag = ...

unionBag :: Bag a -> Bag a -> Bag a
```

(2) Next, we customize our translation with edits, such as

```
rename type GHC.Num.Int = nat
```

(3) Given those, we run `hs-to-coq` and get runnable Coq output, such as

```
Definition unionBags {a} : Bag a -> Bag a -> Bag a := ...
```

(4) We then need to figure out what a specification for the code looks like, which may involve defining custom operations or predicates such as

```
Fixpoint well_formed_bag {A} (b : Bag A) : bool := ...
```

[3] To make it precise, we would have to specify the structure – a homomorphism with respect to the semigroupoidal structure of concatenation, in this case, although we do not prove associativity.

(5) Once we have that model, we can finally sit down and prove desirable theorems (possibly in terms of lemmas), just as in any other Coq development, such as

```
Theorem unionBags_wf {A} (b1 b2 : Bag A) :
  well_formed_bag b1 -> well_formed_bag b2 ->
  well_formed_bag (unionBags b1 b2).

Theorem unionBags_ok {A} (b1 b2 : Bag A) :
  bagToList (unionBags b1 b2) = bagToList b1 ++ bagToList b2.
```

(6) Once all of these theorems are stated and proved, we have a complete verification! At least, until our scope grows...

CHAPTER 3

`hs-to-coq`: Design and Usage

In this chapter, I discuss the design and usage of `hs-to-coq`. Over time, the design goals of `hs-to-coq` have grown, and its aspirations have progressed from mimicking the source code directly (Chapter 5), to attempting to change the source code in order to preserve semantics (Chapter 6), to changing the semantics of the source in order to produce a Coq *model* of the input (Chapter 7). Through all of this, the core of `hs-to-coq` has remained the same: faithful translation of a subset of Haskell to semantically equivalent Gallina, mediated by a collection of *edits*, textual instructions on how `hs-to-coq` should vary the translation. These edits comprise a huge portion of the effort in the `hs-to-coq` code base, and are what allow it to be used in these three different ways (and more – see Section 6.2).

3.1. How we've used `hs-to-coq`

Over the lifetime of `hs-to-coq`, we have adjusted the methodology for using it. At first, our expectation for using `hs-to-coq` was that "Total Haskell is reasonable Coq" (Spector-Zabusky et al., 2018), as discussed in Chapter 5: if we applied `hs-to-coq` to *total* Haskell code, we would get a Coq model that we could reason about directly in the usual ways. This way of using `hs-to-coq` works surprisingly well, and a surprising amount of Haskell fits into it.

Unsurprisingly, though, "a surprising amount" isn't "everything we care about". Our next approach to working with `hs-to-coq` was to use it to produce a translation of code with a total *interface*, even if the implementation used partial techniques (Breitner et al., 2018). This also required working with alternative approaches to termination proofs, as we wanted to handle code that terminated but was not structurally recursive. As we discuss in Chapter 6, this technique enabled us to verify much larger codebases, such as the `containers` library.

The current culmination of our approach, however, takes an even more aggressive approach towards its Haskell input. This approach assumes that we want to verify small portions of a much larger code base, even if those functions are partial, or depend on functions we can't or don't want to verify, or anything else. We have given `hs-to-coq` the ability to apply dramatic changes to its input, and used this to verify a portion of GHC itself (Spector-Zabusky et al., 2019). This technique, discussed in Chapter 7, takes a very different approach than we had originally envisioned, being more comfortable with edits, with axioms, and with partiality. And yet it has perhaps the most promise for working with the Haskell codebases that are used in practice.

In fact, our users[4] are happy with this increased flexibility. Verification of the entirety of large codebases is a daunting task, and trying to get an initial handle on them is not easy – often one finds oneself pulling at something one thinks is a small thread only to find that it goes all the way through the entire codebase. What edits provide is the ability to trim that thread, or identify shorter ones, or even more radical changes. Because they make it easier to begin verifying some portion of the eventual target, edits make using `hs-to-coq` for verification much more *accessible*.

With their different advantages and disadvantages, none of these approaches *replaces* the others – each one builds on the previous, and includes its techniques. Even in GHC, there are plenty of functions that are handled without extra intervention, under the "Total Haskell is reasonable Coq" approach. But as we have expanded our remit to include more and larger codebases, we have learned that we need to power up the abilities of `hs-to-coq`, and allow ourselves to make tradeoffs between the soundness of the verification and the scope of the code we can consider.

3.2. Desiderata

When building `hs-to-coq`, there were a number of considerations that informed its structure. These considerations derived from one overarching goal: to build a trustworthy verification tool that could handle GHC optimization passes. Thinking about trustworthiness, GHC, and the needs of verification, we were guided, informally, by the following five principles:

- The output must be visibly similar to the input (*legibility*).
- The user must never need to edit Haskell code.
- The input and the output must have the same behavior, absent specific requests otherwise.
- Use inductive reasoning, not coinductive (in other words, pretend Haskell is strict.)
- Support real Haskell code – in particular, the sort of Haskell code that is found in GHC.

These principles were not (for the most part) articulated explicitly; they are my own reflections about the ways we were open to extending the design and the ways we knew we didn't want to extend the design.

3.2.1. Legibility. Sadly, it is infeasible to verify `hs-to-coq` itself, for a variety of reasons:

- We lack a formal semantics of Haskell against which to validate the translation.
- We cannot use `hs-to-coq` to verify `hs-to-coq`, as `hs-to-coq` uses more features than `hs-to-coq` supports.
- Put simply, `hs-to-coq` is just too large. If we want to translate as much of Haskell as possible, the amount of work it would take to verify `hs-to-coq` would be prohibitive. Moreover, `hs-to-coq` pulls in a lot of dependencies – it uses Happy (Gill, Marlow, and other contributors, 2010) to generate code, it uses the `lens` library (Kmett, 2018) extensively, and so on.

[4]Including us ourselves.

- The (notional) correctness theorem for `hs-to-coq` is (or would be) both complicated and vague.
  Complicated: Haskell is partial and lazy; Coq is total and focused on inductive types. Haskell has coherent type classes and powerful type inference; Coq has type classes without these guarantees and weaker type inference. The desired correctness theorem is very subtle – it is not simply "the original and translated terms compute to the same result".
  Vague: Without any edits, the Haskell and Gallina programs are expected to be identical. As soon as edits appear, however, this identity is broken, and the desired result is less obvious. How do we specify the correctness of a term that has been the object of a `rewrite` edit?

Consequently, one important design goal of `hs-to-coq` is *legibility*: the Gallina arising from translation must be visibly similar to the original Haskell. This is important because it means that the user of `hs-to-coq` can, and often *will*, compare the input and the output for at least moderate similarity. "Can" is clear, but why "will"? "Will" because during the verification, a programmer will often – though not always – look at both the Haskell and the Coq, and thereby compare the two automatically. This is not as powerful as real verification, and we have a test suite (see Section 3.3) as well, but it does help give credence to our translation and our approach in general.

Additionally, during verification, the user will be staring intently at the translated Coq code, catching obvious bugs in translation; if the user can then successfully verify the desired theorem, this makes it more likely – although not guaranteed! – that the translation was correct. As many correctness theorems depend on all the details of the translated code, it will often be the case that bugs will disrupt the ability to prove a theorem correct. This phenomenon is not unique to `hs-to-coq`, but it works to our advantage here as it does elsewhere.

3.2.2. No manual editing. We want to ensure that `hs-to-coq` enables the user to perform all their verification work without ever editing a Haskell file. This is a key design consideration based mostly around one key observation: *as soon as you edit the input, you stop keeping track of updates to it.* When you've made many manual edits to a code base for the sole purpose of verification, incorporating changes that other people have made for functionality becomes somewhere between difficult and effectively impossible.

Another benefit is that edits are *semantically meaningful*. Suppose that you need to make a change to a Haskell code base so that Coq can support it: for example, renaming a punned constructor (as in `newtype Identity = Identity a` – the first `Identity` is the type name, and the second is the constructor name) everywhere it occurs. When looked at in a textual diff, it is impossible to see what change was made, especially if *multiple* changes have been made at the same time. If, instead, you read the edit `rename value Con = Mk_Con`, you know exactly what change was made, and can understand it, revert it, bring it into a new code base, or whatever else you need to do.

The culmination of all these benefits is that the design of `hs-to-coq` enables a *mechanized formalization gap*, a machine- and human-readable record of the changes between the artifact and the model. (For further discussion, see Section 3.4.)

Note that we do *not* get as a benefit that the user does not need to know Haskell. Figuring out the edits to make still requires being able to read and understand the Haskell code base. If the work is divided properly, there can be people working solely on the proofs who know only Coq (as has sometimes been the case in our work), but there needs to be some Haskell expertise on the team.

3.2.3. Same behavior. The only way for `hs-to-coq` to be useful is for the input Haskell and the output Coq to have the "same" behavior. But "same" covers a fairly broad spectrum of similarities. At their most basic, Haskell and Coq are both lambda calculi with algebraic data types; this gives them a solid core of overlap in behaviors to start with, and is the foundation on which `hs-to-coq` is based. Functions translate to functions, data types translate to data types, and so β-reduction translates to β-reduction. But despite this, there are some behaviors of Haskell that just don't translate to Coq, both small scale and large scale. At the small scale, we have features such as pattern-matching with guards and fall-through; Haskell's pattern language is stronger than Coq's, so it requires some desugaring. At the large scale, some features of Haskell are simply incompatible with Coq, chief among them nontermination.

Both these sorts of features must be translated somehow in `hs-to-coq`, although the story is different for the two of them. For small-scale features, we must decide on a desugaring that preserves the semantics, while not totally transforming the code (see Section 3.2.1). For pattern matching, this means translating guards into if statements, and creating local functions containing a suffix of the branches to translate fall-through. This preserves the semantics of pattern matching exactly.

For nontermination, we need to make harder choices: how can we write a program in a terminating language that's the "same" as one with nontermination? We must make choices, and decide what level of "sameness" is acceptable. Edits allow us some flexibility in this choice; for instance, we can `rewrite` nontermination away, or use `termination` to demonstrate that something was actually terminating even though Coq couldn't see it. Our most powerful technique is the `Default` type class, which provides inhabited types with opaque default values. (For more on this technique, see Section 3.8.) This allows us to translate explicit calls to crashing functions to this opaque value, a correspondence which in practice we have found to be close enough; however, when making this translation, we sacrifice the ability to provide termination guarantees for the original Haskell code.

One concession we make to the difference between Haskell and Coq is around laziness: absent specific requests otherwise, we pretend that Haskell is strict. The choice of what to do here was so important it formed an entire principle of our design, which we discuss next.

3.2.4. Induction, not coinduction. In a seeming contradiction, another design goal was to use only inductive reasoning, and ignore the fact that Haskell is lazy and fundamentally coinductive. Nevertheless, this has worked well in practice; why? Our initial rationale for this choice was twofold:

(1) Many Haskell programs *are* inductive, and in particular, compilers (such as GHC) operate on finite syntax trees.
(2) Coinductive reasoning is much harder, especially out of the box.

And indeed, these two principles have remained true.

Another complication is that, even some Haskell data types that *are* used coinductively are often *also* used inductively. For example, consider lists: the standard library provides both `length`, which assumes lists are inductive, and `repeat`, which assumes lists are coinductive.[5] (This goes back to the discussion in Section 2.3 – sometimes programmers use one type, such as `[]` or `Int`, for multiple purposes, and we are more likely to need to tease those apart in Coq.) So we cannot strictly assume that types are coinductive instead. In practice, inductive reasoning seems to be the most common, and so defaulting to that has served us well. But we did eventually provide support for specifying that certain types or function ought to be treated coinductively (Section 4.1.7).

3.2.5. Support real Haskell code. The motivation for building `hs-to-coq` was originally tied up with the question of verifying parts of GHC. This meant that designing `hs-to-coq` to work with a subset or a dialect of Haskell was always out of the question: it had to support anything GHC could throw at it. This has had two ramifications, pushing in opposite directions. First, it's made `hs-to-coq` a general-purpose tool: GHC is *so big* that it uses essentially every basic feature of Haskell (that is, the features contained in Haskell 2010), and so `hs-to-coq` has to support all of those. But second, GHC itself only uses certain extensions – for instance, it uses both `CPP` and `NamedFieldPuns`, but does not use `TypeApplications` or `ApplicativeDo` – and so we added support for features beyond Haskell 2010 lazily. Each time one came up – in GHC or, later, elsewhere – we would extend `hs-to-coq` as necessary to support it. But the base was always "all of Haskell 2010", and this combined with our principle of the user never needing to edit Haskell files (Section 3.2.2) meant that we had to choice but to grow `hs-to-coq` to support more and more varieties of code.

3.3. Test suite

One way that we validate `hs-to-coq`'s behavior is with our test suite. We have a collection of Haskell examples and edit files, along with a Makefile that can automatically translate the examples into Coq and typecheck those Coq files. These files are known to be good; if they ever fail to translate and type check, we know we have introduced a bug. These examples are all bite-sized (the largest is 45 non-blank non-comment lines), and thus serve as a way to guard that individual edits or other features of `hs-to-coq` are behaving correctly. In addition, we have a Travis CI server (Travis CI, GmbH, 2020) set up for continuous integration testing; every push to our repository triggers a build of `hs-to-coq`, including running tests, so that we can ensure that they continue to pass.

[5] While `length` could be defined for infinite lists if it returned a conatural number, it returns an `Int`.

3.4. Mechanized formalization gaps

Because `hs-to-coq` is not itself verified, it is the most obvious source of a formalization gap between the Haskell input and the Coq output. But one of the key contributions of this work is that `hs-to-coq` enables a *mechanized formalization gap* between the input and the output. The idea here is that `hs-to-coq` produces a Coq model of the Haskell code, but with certain key differences. Traditionally, these difference would be ad-hoc, produced by hand by editing the source or the model. What `hs-to-coq` provides is a *record* of the changes – the edits, the preamble, and the midamble – which can be *mechanically interpreted* along with the input to repeatably produce the same model. This means that our formalization gap is *itself* an artifact that can be read by humans or used and analyzed by machines.

3.5. Infix operators

In addition to philosophical considerations, we also need to look at the details of how `hs-to-coq` translates code in order to be able to read the output. One such detail that shows up in many code samples is how `hs-to-coq` represents Haskell infix operators in Coq. While both Haskell and Coq support infix operators, they handle them very differently. In Haskell, `(+)` is just as good a name as `add`; both can be bound, exported, referenced as a qualified name, etc. The only difference is that symbolic names `(+)` can be used infix without its parentheses, and ordinary alphanumeric names like `add` can be used infix when surrounded by backticks. In Coq, on the other hand, infix operators are solely surface-level syntax, defined through Coq's powerful `Notation` mechanism (or its abbreviated form `Infix`); an expression like `m + n` can be defined to translate into `add m n`, and the two are then identical in every way after parsing. However, `+` does not have any sort of independent existence, and is only available if the notation has been brought into scope; there is no way to reference notations with a module qualifier.

This distinction poses multiple challenges for `hs-to-coq`. The first and simplest is, what automatic names do we give to operators? GHC has a scheme called *z-encoding*[6] for representing operators in contexts (such as linkers) where only C-like names are allowed: all symbols are replaced with a "`z`" or a "`Z`" followed by a mnemonic letter or Unicode escape sequence. So, for example, `&&` becomes `zaza`; the "`a`" stands for ampersand. As an example of the Unicode translation scheme, ∘ (which we use to translate the composition operator `.`) becomes `z2218U`, since that character is U+2218 RING OPERATOR; the encoded form is the hexadecimal representation of the Unicode code point followed by a `U`. We use this encoding scheme to translate names; an infix name gets translated to `op_`, followed by the z-encoding of the name, followed by `__`. So continuing the first example, `(&&)` becomes `op_zaza__`.

Now, in order to reference the name, `hs-to-coq` provides Coq notation to avoid this ugly form. When translating code, `hs-to-coq` defines both the infix operator notation `b1 && b2`, as well as a "prefix" notation `_&&_` that can be used anywhere a variable can; this stands in for Haskell's `(&&)` form, and is used in operator sections. This is achieved with generated code like the following:

[6] https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/symbol-names

```
Notation "'_&&_'" := (op_zaza__).
Infix "&&" := (_&&_) (at level 99).
```

However, `hs-to-coq` refers to all names qualified by default, rather than trying to manage imports itself. That would limit us to using the names of the form `Some.Mod.op_zaza__`, which is unpleasant. Thus, whenever we generate notations for a symbolic name, we *also* generate *manually qualified* forms of those notations:

```
Notation "'_Some.Mod.&&_'" := (op_zaza__).
Infix "Some.Mod.&&" := (_&&_) (at level 99).
```

The `Some.Mod` in these names isn't functioning as a module name; it's just part of the syntax of these notations. But by using the module name as part of that syntax, we ensure that this notation is globally unique, which means that we can import it and bring it into scope whenever we want. Thus, the full translation is to wrap those last two lines in a module called `Notations`, unlike the unqualified forms:

```
Notation "'_&&_'" := (op_zaza__).

Infix "&&" := (_&&_) (at level 99).

Module Notations.
Notation "'_Some.Mod.&&_'" := (op_zaza__).
Infix "Some.Mod.&&" := (_&&_) (at level 99).
End Notations.
```

Then, whenever `hs-to-coq` generates a `Require Some.Mod.`, it will follow that up with `Import Some.Mod.Notations.`, allowing us to fake having qualified operators.

3.6. Notation for literals

Another place we introduce extra notation in our translation is when translating literal numbers, strings, and characters. In Haskell, integer literals such as `42` are syntactic sugar for `fromInteger (42 :: Integer)`, with the latter `42` being a putative non-overloaded integer. (Our translation doesn't handle fractional literals, but Haskell interprets them analogously.) Coq, on the other hand, handles integer literals rather differently: it provides support for overloading them via its powerful notation system, but only by writing a plugin in OCaml. Instead of needing to distribute an OCaml plugin, we instead provide a lightweight notation for `fromInteger` in our handwritten `GHC.Num` module:

```
Notation "'#' n" := (fromInteger n) (at level 1, format "'#' n").
```

This notation binds tightly and is formatted without a space, letting us write `#42`. We still indirectly leverage Coq's notation system here: since `fromInteger` takes an `Integer` as input, its Coq translation takes a `Z`, and Coq will use that type information to decide to interpret the integer literal as a `Z` no matter what the current notation scope is. We make this notation available both from `GHC.Num` and from `GHC.Num.Notations`, ensuring that `#` will always be available to translated programs just like qualified infix operators.

We then set up `hs-to-coq` to introduce references to `#`; helpfully, the abstract syntax tree that GHC provides represents Haskell's overloaded numeric literals as applications of `GHC.Num.fromInteger` to non-overloaded numeric literals. We were thus able to add a special case in `hs-to-coq`'s pretty-printer to detect applications of this form and output them using `#`. This way, in combination with our support for infix operators, we can translate Haskell expressions such as `40 + 2 == 42` into Coq expressions such as `(#40 GHC.Num.+ #2) GHC.Base.== #42`. The only catch is that, while `hs-to-coq` will not print out `#` for general (non-literal) uses of `fromInteger`, Coq's notation system is not so forgiving; any ordinary use of `fromInteger` in a Haskell program will, once translated with `hs-to-coq`, show up as `#` in a proof state or if otherwise printed out by Coq, even though the `hs-to-coq`-generated file will use the original function name.

The situation with string literals is similar, but less streamlined. We do not support the `OverloadedStrings` language extension, which simplifies our job somewhat; Haskell string literals are thus simply syntactic sugar for lists of characters. However, once again, without writing an OCaml plugin, we are limited to existing interpretations of strings. The only default such interpretation is as a member of the type `Coq.Strings.String.string`, which is is isomorphic (but not equal) to a list of ASCII characters, each of which is represented by eight `bool`s. We do not worry about Unicode for now, but we still have to provide a conversion function from Coq `String`s to Coq `list`s; we call this function `GHC.Base.hs_string__`, and define it in the midamble for `GHC.Base`. We similarly have to define a translation from Coq `Coq.Strings.Ascii.ascii`s to our translated `Char`s; we call this function `GHC.Char.hs_char__`.[7] We then provide a short prefix notation for each of these functions, like `#` above: `&` for strings and `&#` for characters. This enables us to write strings that look like `&"hello, world"` and characters that look like `&#"*"` (a single character in double quotes being Coq's syntax for `ascii` values).

This is not as streamlined as the situation for numeric literals, however, for the following three reasons. First, `hs-to-coq` will introduce explicit references to `GHC.Base.hs_string__` and `GHC.Char.hs_char__` rather than use the `&` and `&#` notations. Second, if any string or character literal occurs in a translated module, the user must manually `Import` the appropriate module in the preamble in order for Coq to be able to parse it (for strings, the module `Coq.Strings.String`, or at least its submodule `StringSyntax`; for characters, the module `Coq.Strings.Ascii`, or at least its submodule `AsciiSyntax`). And third, the `&` and `&#` notations are not exported in a `Notations` module, so you must `Import GHC.Base` and/or `GHC.Char` in order to get access to the notations. (Just importing `GHC.Base` is sufficient for both, since it reexports `GHC.Char`.)

The restrictions around the notation names are minor limitations of `hs-to-coq`. The requirement to manually import the string and character syntax is harder to eliminate, since string literals do not include references to the `Coq.Strings.String` module that can be picked up by `hs-to-coq`'s dependency tracking (unlike the function `GHC.Num.fromInteger`, which does implicitly). Regardless, since we do not prove

[7] We represent Haskell `Char`s as `N`s, Coq's type of binary natural numbers, meaning this function is simply `Coq.Strings.Ascii.N_of_ascii`.

very much about strings, these are not terrible limitations, and could be lifted as part of future work. Additionally, the same phenomenon that makes Coq print out `#` works in our favor here: because Coq applies notations when printing terms, `&` and `&#` are printed out by Coq once they are in scope. This means that when using Coq to print definitions, or when examining a proof state, literal strings will generally appear in the desirably concise manner we might hope for.

3.7. Transforming code automatically

Another detail we need to understand about `hs-to-coq` is how and when it transforms the input Haskell code without edits. This does happen; although legibility is an important desideratum, Coq is sufficiently dissimilar from Haskell that maintaining it requires careful thought. For example, Haskell's pattern-matching language is syntactically rich, supporting a variety of features: equational definitions; nested patterns; and both boolean and pattern guards with fallthrough semantics. Coq, on the other hand, supports only `match ... with ...` and `if ... then ... else ...` expressions, with the former supporting nested patterns. This means that we need to rearrange the structure of complex Haskell patterns to be compatible with Coq, which does require breaking legibility somewhat; this is unavoidable, since the fall-through semantics of Haskell patterns are simply not available in Coq. We also ensure that we detect when patterns *can* be translated more directly and do so, which is why the examples we have seen so far have all looked similar when translated.

A more complex example of this is how `hs-to-coq` translates type classes. By default, `hs-to-coq`'s translation of type classes is somewhat surprising. A simple type class and instance such as

```
class C a where
  m :: a
  f :: a -> Bool

instance C () where
  m   = ()
  f _ = True
```

is not simply converted to a standard Coq type class, but instead to the following code where it has been encoded in a continuation-passing style (or *CPSed*):[8]

```
(* class C a where ... *)

Record C__Dict a := C__Dict_Build {
  f__ : a -> bool ;
  m__ : a }.

Definition C a :=
```

[8]We also see that the order of the methods changes; this is because `hs-to-coq` topologically sorts definitions in dependency order, and is not guaranteed to preserve the order in the source file when doing so.

```
  forall r__, (C__Dict a -> r__) -> r__.
Existing Class C.

Definition f `{g__0__ : C a} : a -> bool :=
  g__0__ _ (f__ a).

Definition m `{g__0__ : C a} : a :=
  g__0__ _ (m__ a).

(* instance C Bool where ... *)

Local Definition C__unit_f : unit -> bool :=
  fun arg_0__ => true.

Local Definition C__unit_m : unit :=
  tt.

Program Instance C__unit : C unit :=
  fun _ k__ => k__ {| f__ := C__unit_f ; m__ := C__unit_m |}.
```

Here, the generated Coq code defines the simple record encoding of the type class as `C__Dict`, whose double-underscore name indicates that it is a generated name from `hs-to-coq`; the fields of that dictionary look like the methods of `C`, but have trailing double-underscore names. The type class `C` itself is a universally-quantified CPS wrapper around `C__Dict`, and the methods `m` and `f` "unwrap" it by extracting the fields. The instance is then put together not by defining a record of `Local Definition`s, but by defining a function that passes such a record to a continuation.

Even after making all these changes, legibility still matters: as we see, the transformation is limited to the definition site where we put together the type class. Both the method definitions and, more importantly, the call sites of the methods are unchanged.

We do not simply perform this complex translation for fun, of course – the importance of this feature is that it allows Coq to automatically reduce expressions containing a type class much more smoothly and uniformly, which noticeably eases our proofs. However, it is also much more involved, which, in addition to being an aesthetic nuisance (and occasionally requiring extra unfolding in proofs), actually gets in the way of encoding more complex Haskell type classes that contain associated types. We can avoid this last problem by shutting off this transformation with the **`simple class`** edit (discussed in detail in Section 8.5.2), but the costs are still real. The tradeoffs this transformation imposes, as we can see by looking at the output, highlight two important things: some of the reasons we might need to produce a less legibile transliteration, and some of the reasons why legibility is such an important desideratum.

3.8. Partiality

Sometimes, unfortunately, Haskell code is partial. How does `hs-to-coq` handle that? One intuitively-appealing approach would be to translate partial code into an "error monad" of some sort. Let's suppose that we have

```
Definition ERR (a : Type) : Type := string + a.
```

Then we would translate

```
head :: [a] -> a
head []    = error "Prelude.head: empty list"
head (x:_) = x
```

into

```
Definition head {a} : list a -> ERR a :=
  fun arg_1__ =>
    match arg_1__ with
      | nil => inl "Prelude.head: empty list"
      | cons x _ => inr x
    end.
```

However, this would pose a great many difficulties. For one thing, because Haskell-functions are curried, it becomes tricky to see where we insert the error monad. Does

```
(++) :: [a] -> [a] -> [a]
```

become

```
Definition op_zpzp__ {a} : [a] -> ERR ([a] -> ERR [a]) := ...
```

or

```
Definition op_zpzp__ {a} : [a] -> [a] -> ERR [a] := ...
```

and how do we know which? The former is the only uniform option, but then translating function calls becomes ugly: translating

```
head ((++) xs ys)
```

would have to become the Coq equivalent of

```
head =<< ((++) xs >>= ($ ys))
```

and the implementation of `(++)` would have to become similarly heinous.

One might hope to mitigate this issue by only annotating those functions that require partiality, and keeping track of this inside `hs-to-coq`. Then `head` would still have the type `forall {a}, list a -> ERR a`, but `(++)`/`op_zpzp__` would retain its original type of `forall a, list a -> list a -> list a`. Unfortunately, this starts to struggle when presented with higher-order functions, such as

```
map :: (a -> b) -> [a] -> [b]
```

To be maximally general, its argument would still need to be translated as a potentially-partial function, and thus its result would need to be partial:

```
map : forall {a} {b}, (a -> ERR b) -> [a] -> ERR [b]
```

This is true even though `map` is *itself* total! We could try to work around this by duplicating the definition of `map` and defining both partial and total translations, but this now poses the problem of duplicated code that is nearly identical.

A translation like the one outlined above is entirely reasonable in many contexts: theoretical reductions, compilation procedures, and so on. Indeed, Abel, Benke, Bove, Hughes, and Norell (2005) used such a technique, combined with abstracting over the monad, in a translation from GHC's internal language to an early version of the Agda language (for a longer discussion, see Section 9.2.3.3). However, our use case requires a programmer to work with the translated code, and to be able to integrate it with existing Coq libraries. Program transformations like this make that incredibly difficult, and increase the mental friction when comparing the source and target languages. And this transformation is relatively simple: it can affect every arrow exactly once, and the term-level change is simply to put the program into monadic form. The lesson is once again, just as we saw in Section 3.7, that code transformations must be kept to a minimum: they can *happen*, but they need to preserve as much of the code's structure as possible.

Instead, we focus on our desideratum of *legibility* (see Section 3.2 and Section 3.2.1). We need to make sure that however we handle partiality, it produces *readable code*. And the most readable code is code that is *total*, not partial. So what if we assume *all* Haskell code is total? Our original design goal was to only work with total Haskell code (Chapter 5), although this quickly became untenable; nevertheless, *so much* Haskell code is total that this still works well much of the time.

Thus, `hs-to-coq` doesn't handle partiality with any sort of clever encoding; it simply plugs its ears and pretends it doesn't exist. Some good fraction of the time, this gets us far enough. And it can never go wrong – any code that's *actually* partial, such as `head` or `loop = loop`, will simply cause the compilation of the Coq code to fail.

But sometimes we need to *actually* deal with partial code. Whether internal to functions (as we will see in Chapter 6) or at the top level (as we will see in Chapter 7), real Haskell code uses partiality, and we need to handle it.[9]

The approach `hs-to-coq` takes, which was developed by my collaborator Joachim Breitner, is to use a type class called `Default`. `Default` is a type class for inhabited types, and it contains a single method, `default`:

```
Class Default (a : Type) := {
  default : a
}.
```

The important thing about this type class is that we can use it to define `error` and `undefined` safely: if their return types are constrained by `Default`, then they can be total. However, this could introduce bugs where we rely on the particular *value* of `Default`. So when we define these terms, we make them *opaque*:

```
Definition error {a} `{Default a} : String -> a.
```

[9] None of this touches on recursion; for that, see Section 3.9.

```
Proof. exact (fun _ => default). Qed.

Definition undefined {a} `{Default a} : a.
Proof. exact default. Qed.
```

Because we use `Qed`, Coq cannot expand the definitions of `error` and `undefined`. Thus, any proof that runs across them can never rely on their exact value, preventing us from causing bugs due to overdefinition of partial functions. In fact, beyond trivial proofs – such as `eq_refl : undefined = undefined :> a` – we cannot prove anything about `undefined`, and so any nontrivial proof that invokes it accidentally will get stuck.

However, this is still not the same as having a true bottom value ($\bot$), for two reasons. First, Haskell functions must be *continuous* with respect to $\bot$ (shenanigans with `IO` notwithstanding), and in general it is possible to say what the result of a function applied to $\bot$ is (in a mathematical sense, not from within Haskell). In Coq, we have no such continuity guarantees, and attempting to evaluate pattern-matches on `default` gets stuck. Second, in Haskell, $\bot$ is a unique value distinct from all other values of a type. On the other hand, `default` behaves like an *unknown* value selected from the total type. In Haskell, the type `()` has two values, `()` and $\bot$; in Coq, `unit` only has one value, and we can prove that `default = tt`.

3.9. Recursion

The previous section shows how `hs-to-coq` takes care of direct partiality, but that's not the only source of bottoms to worry about. Haskell also supports general (co)recursion, and Coq decidedly does *not*. Sources of genuine unbounded recursion (and dually, of unbounded nonproductive corecursion) are bugs – there is no merit to trying to place these in a form Coq can handle. But Coq's termination (and guardedness) checkers are incredibly conservative, and rule out many terminating Haskell functions (or guarded Haskell covalues). There are four cases to worry about:

(1) Structural recursion, which Coq accepts;
(2) Nonstructural but terminating recursion, which Coq needs help accepting;
(3) Guarded corecursion, which Coq could accept if it weren't being misinterpreted as recursion; and
(4) Nonguarded corecursion.

Since we have decided to focus `hs-to-coq` on recursion and not corecursion (Section 3.2.4), we have correspondingly deemphasized Items 3 and 4. We provide the `termination corecursive` edit, which marks values as corecursive, and the `coinductive` edit, which marks types as coinductive (Section 4.1.7), but this is as far as we go – we do not provide support for Item 4, going beyond the guardedness checker, and in practice this has never been an issue. In future work, we could extend our Coq tooling to support existing techniques for working with coinduction, such as the Paco library (Hur, Neis, Dreyer, and Vafeiadis, 2013).

Item 1, structural recursion, occurs very frequently; our first paper, "Total Haskell is reasonable Coq" (Spector-Zabusky et al., 2018), dealt almost exclusively with such functions, and they occur even in codebases as large as GHC (Spector-Zabusky

et al., 2019). But it is not, of course, the be-all and end-all of recursion, and Haskell programmers often write functions that depend on clever termination arguments. There is, by and large, not anything to say about how structural recursion works. Sometimes, however, it doesn't pop out immediately; it can be necessary to rewrite functions in small ways to get the structural recursion to be recognized; for example, replacing functions that operate on lists with simpler functions and separate calls to `map`. This is more common when working with mutual recursion (see Section 3.9.2), however, since we can instead rely on `hs-to-coq`'s support for nonstructural recursion when working with single recursive functions.

In "Ready, `Set`, verify! Applying `hs-to-coq` to real-world Haskell code (experience report)" (Breitner et al., 2018), we divided the different kinds of terminating recursion into the following categories:

(1) Structural recursion
(2) Almost-structural recursion
(3) Well-founded recursion
(4) Deferred recursion

We can also add a few more categories that we can handle:

(5) Structural mutual recursion
(6) Nearly-structural mutual recursion
(7) Corecursion

Each of these categories requires different techniques to convince `hs-to-coq` to accept them; we leave off those categories that `hs-to-coq` cannot accept.

3.9.1. Structural recursion. The simplest case: do no work. Structural recursion corresponds to the `fix` keyword in Coq, and arises in cases from toy examples all the way to GHC. A function like `map` is a paradigmatic example:

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

This is the definition of `map` that's used in the GHC `Prelude`, and it translates to

```
Definition map {a b} : (a -> b) -> list a -> list b :=
  fix map arg_1__ arg_2__ :=
    match arg_1__, arg_2__ with
      | _, nil       => nil
      | f, cons x xs => cons (f x) (map f xs)
    end.
```

This implementation of `map` is structurally recursive and is accepted by Coq; the call to `map` occurs on `xs`, which is a strict subterm of `arg_2__`.

3.9.2. Mutual recursion. Mutual recursion is nominally the same, but in practice presents its own issues. Coq's idea of what counts as structural recursion is very strict – more so than what Haskell programmers seem to think of – and nonstructural

recursion is not supported by **Program Fixpoint**. So when working natively with Coq, we need to provide ways to work with its definition of structural mutual recursion.

For an example of how this works,[10] consider the following pair of mutually recursive Coq data types, which represent a `Forest` of nonempty `Tree`s. Each `Branch` of a `Tree` holds an extra boolean flag, which we can extract with `isOK`.

```
Inductive Forest a : Type
  := Empty    : Forest a
  |  WithTree : Tree a -> Forest a -> Forest a
with Tree a : Type
  := Branch : bool -> a -> Forest a -> Tree a.

Arguments Empty    {_}.
Arguments WithTree {_} _ _.
Arguments Branch   {_} _ _ _.

Definition isOK {a} (t : Tree a) : bool :=
  match t with
  | Branch ok _ _ => ok
  end.
```

Now, if we ignore this boolean flag, we can write a pair of mapping functions to apply a function to every value in a `Forest` or a `Tree`, and this works fine:

```
Fixpoint mapForest_always
    {a} (f : a -> a) (ts0 : Forest a) {struct ts0} : Forest a :=
  match ts0 with
  | Empty         => Empty
  | WithTree t ts => WithTree (mapTree_always   f t)
                              (mapForest_always f ts)
  end
with mapTree_always
    {a} (f : a -> a) (t : Tree a) {struct t} : Tree a :=
  match t with
  | Branch ok x ts => Branch ok (f x) (mapForest_always f ts)
  end.
```

However, suppose the boolean represents some validity check (in real code, this sort of thing would be more complicated). So we check the boolean first, and only actually recurse if necessary. This means we have *three* mutually recursive functions, but one of them is just handing the work along. Unfortunately, this doesn't work:

```
Fail Fixpoint mapForest_bad
    {a} (f : a -> a) (ts0 : Forest a) {struct ts0} : Forest a :=
```

[10] I originally presented this example on Stack Overflow: "Can I do 'complex' mutual recursion in Coq without let-binding?", https://stackoverflow.com/q/52599324/237428.

```
  match ts0 with
  | Empty         => Empty
  | WithTree t ts => WithTree (mapTree_bad   f t)
                              (mapForest_bad f ts)
  end
with mapTree_bad
    {a} (f : a -> a) (t : Tree a) {struct t} : Tree a :=
  if isOK t
  then mapOKTree_bad f t
  else t
with mapOKTree_bad
    {a} (f : a -> a) (t : Tree a) {struct t} : Tree a :=
  match t with
  | Branch ok x ts => Branch ok (f x) (mapForest_bad f ts)
  end.
```

The problem is that `mapTree_bad` calls into `mapOKTree_bad` on an argument that isn't actually smaller.

Except... all `mapOKTree_bad` is doing is an extra step after some preprocessing. This *will* always terminate, but Coq can't see that. To persuade the termination checker, we can instead define `mapOKTree_good`, which is the same but is a local `let`-binding; then, the termination checker will see through the `let`-binding and allow us to define `mapForest_good` and `mapTree_good`. If we want to get `mapOKTree_good`, we can just use a plain old definition after we've defined the mutually recursive functions, which just has the same body as the `let`-binding:

```
Fixpoint mapForest_good
    {a} (f : a -> a) (ts0 : Forest a) {struct ts0} : Forest a :=
  match ts0 with
  | Empty         => Empty
  | WithTree t ts => WithTree (mapTree_good   f t)
                              (mapForest_good f ts)
  end
with mapTree_good
    {a} (f : a -> a) (t : Tree a) {struct t} : Tree a :=
  let mapOKTree_good {a} (f : a -> a) (t : Tree a) : Tree a :=
        match t with
        | Branch ok x ts => Branch ok (f x) (mapForest_good f ts)
        end in
  if isOK t
  then mapOKTree_good f t
  else t.

Definition mapOKTree_good {a} (f : a -> a) (t : Tree a) : Tree a :=
```

```
  match t with
  | Branch ok x ts => Branch ok (f x) (mapForest_good f ts)
  end.
```

Sadly, Coq can't do this automatically, and it requires duplicating code. Happily, since we're writing a translator, we *can* handle it, and have the translator duplicate code for us: the edit `inline mutual func` will cause `func` to be treated like `mapOKTree_good` in the example above. Although the code is duplicated, the local and global functions named `func` are judgementally equal, and so can be converted between.

For example, we could produce the `_good` family of functions, dropping the suffix, from the Haskell code

```
data Forest a = Empty
              | WithTree (Tree a) (Forest a)

data Tree a = Branch Bool a (Forest a)

isOK :: Tree a -> Bool
isOK (Branch ok _ _) = ok

mapForest :: (a -> a) -> Forest a -> Forest a
mapForest _ Empty           = Empty
mapForest f (WithTree t ts) = WithTree (mapTree   f t)
                                       (mapForest f ts)

mapTree :: (a -> a) -> Tree a -> Tree a
mapTree t | isOK t    = mapOKTree f t
          | otherwise = t

mapOKTree :: (a -> a) -> Tree a -> Tree a
mapOKTree (Branch ok x ts) = Branch ok (f x) (mapForest ts)
```

and the single edit

```
inline mutual mapOKTree
```

3.9.2.1. *Nonstructural mutual recursion.* Nonstructural mutual recursion is perhaps a "white whale" for `hs-to-coq`. While we can handle structural mutual recursion through Coq's native support, and even use edits to hack mutually recursive functions to fit Coq's narrow definition of "structural", true nonstructural mutual recursion lies tantalizingly out of reach. Sadly, `Program Fixpoint` and `Function` simply don't support mutual recursion, so Coq's built-in functionality isn't up to the task. The axioms that `Program Fixpoint` depends on, and the `deferredFix` axiom, are also only given in a form that works with unary functions. It is not impossible to extend these to an n-ary form, but it becomes complicated, and full support requires being to generate the n-ary form systematically. We have been able to get almost far enough

with our edits to support structural mutual recursion, so this is where we have left it for now.

Perhaps our best hope for nonstructural mutual recursion is the EQUATIONS package (Sozeau and Mangin, 2019). This package provides support for a different, more equational way of writing Coq definitions: a definition such as

```
Fixpoint map {a b} (f : a -> b) (xs : list a) : list b :=
  match xs with
  | [::]     => [::]
  | x :: xs' => f x :: map f xs'
  end.
```

would become

```
Equations map {a b} (f : a -> b) (xs : list a) : list b :=
map _ [::]      := [::] ;
map f (x :: xs) := f x :: map f xs.
```

This new syntax is not strictly cosmetic – it also automatically supports dependent pattern-matching and nonstructural recursion, *including* mutual recursion. And it looks more like Haskell, to boot!

Sadly, the prospect of retooling `hs-to-coq` to output EQUATIONS's syntax is nontrivial, and has never been a sufficiently high priority for us that we have been able to devote the time to changing the internals of `hs-to-coq` to work with EQUATIONS. But this remains a promising avenue for future work, that could at once give us more Haskell-looking code and enable more features.

CHAPTER 4

The Edit Language

In this chapter, I discuss the design and structure of the edit language of `hs-to-coq`. Edits are the key novel feature of `hs-to-coq`; they allow for our workflow centered around never modifying code, and they form the mechanized formalization gap that allows us to have a machine-readable record of how the verified artifact differs from the original source. What makes these edits able to serve in these roles is their *breadth* and their *structure*. If we are to avoid editing generated code, we must have a comprehensive ability to make changes; if we are to have a higher-level understanding of the formalization gap, we need edits to be semantic instead of textual. Understanding the panoply of features supported by `hs-to-coq`'s edits is critical to understanding why it is an effective tool.

This chapter presents an overview of and the motivation for the edit language; it discusses both the design of the edit language (Sections 4.1 and 4.2) and how to read edit files (Sections 4.3 and 4.4). For a detailed look at each of the 34 different edits that `hs-to-coq` supports, see Chapter 8.

4.1. The eight categories of edits

For ease of understanding, we can separate the 34 edits supported by `hs-to-coq` into eight broad categories:

- Skipping Haskell code (Section 4.1.1);
- Axiomatizing Haskell code (Section 4.1.2);
- Adding Coq code (Section 4.1.3);
- Changing the structure of the Haskell code (Section 4.1.4);
- Rewriting expressions (Section 4.1.5);
- Providing extra information (Section 4.1.6);
- Proving termination (Section 4.1.7); and
- Meta-edits (Section 4.1.8).

4.1.1. Skipping Haskell code. There's too much Haskell code in the world to verify all of it. But luckily, we're not trying to. Not only are we picking the code bases we want to verify, we're not verifying all the code in them – we usually have a very specific target. And during the verification process, we want to start small and grow our scope. Thus, `hs-to-coq` has a number of edits to support *skipping* pieces of Haskell, from parts of a definition to whole modules. These can be used to skip definitions that we'll verify later, to skip things like `Show` instances that are out of the scope of verification, and so on.

4.1.2. Axiomatizing Haskell code. Skipping work does make a programmer's life easier, but sometimes it causes more problems in the long run. Sometimes, the definitions you skip end up being used in parts of the code you care about. That may be because you want to leave something as part of the trusted computing base; it may be because you don't really want to deal with a function named something like `unsafeInvertBitsAndRewrite`; it may be because you're in the middle of development and would like to put off dealing with this definition for now. For whatever reason, though, sometimes you just want to stub things out. This is why `hs-to-coq` provides a number of edits that support replacing definitions – or even whole modules – with *axioms* instead. These axioms have no computational content, but they allow typechecking and compilation to continue by treating these definitions as black boxes. Axioms can of course introduce unsoundness, even when stubbing out Haskell code (for example, imagine `axiomatize definition GHC.Err.undefined`), so care must be used; however, edits are already part of the trusted computing base in general, so this does not make things any worse, except perhaps slightly in degree.

4.1.3. Adding Coq code. Sometimes, it becomes necessary to write even *more* Coq code. We can do that with the preamble and midamble (see Section 4.2), but sometimes we want to do something slightly smaller or more principled, or in a way that integrates with the rest of the edits.

4.1.4. Changing the structure of the Haskell code. Sometimes, the structure of the Haskell code doesn't play well with Coq. (In some cases, this is `hs-to-coq`'s fault.) Usually, this means that it gets in the way of termination checking. These edits allow us to simplify or adjust the structure of the Haskell code in straightforward ways, so that Coq can work with the result.

4.1.5. Rewriting expressions. Sometimes we need to do finer surgical work. Sometimes, a fragment of Haskell code just won't work in Coq – it can't get past the termination checker, it relies on laziness, or it refers to functions that aren't translated or aren't verified. Or sometimes, there's just a name collision. The most common example of this is a Haskell type and constructor with the same name, which is an incredibly common pattern; as an archetypical example, consider the definition `newtype Identity a = Identity { runIdentity :: a }` in the module `Data.Functor.Identity` from the `base` library. For whatever reason you may need it, `hs-to-coq` provides you with facilities to make these sub-definition edits to your Haskell code.

4.1.6. Providing extra information. Sometimes, `hs-to-coq` gets things *almost* right, but not quite. If only it knew just a bit more – about type annotations, about the correct ordering of definitions, ... That's why `hs-to-coq` provides some edits for teaching it more than it could figure out by itself, or changing its mind over things it did figure out.

4.1.7. Proving termination. One of the biggest differences between Haskell and Coq is, of course, Coq's termination checker. While it's surprising how much Haskell code *is* structurally recursive, or can be made so with minor tweaks (Sections 4.1.4

and 4.1.5), that's not all of it, so `hs-to-coq` naturally provides support for telling the translation to use more powerful forms of termination-checking, or even – to leverage the other huge difference – coinduction.

4.1.8. Meta-edits. A "meta-edit" is an edit that takes another edit as a parameter, allowing the user to control the behavior of any of the other 7 categories of edits we saw above. The only meta-edit in `hs-to-coq` at the moment is `in`, which constrains the scope of any other edit to apply only within the definition of a specific term. This is particularly useful with "big hammer" edits such as those that rewrite terms, which can be written much more clearly if their domain is smaller (so that they don't accidentally apply to unintended parts of the code).

4.2. The history and design of the edit language

We arrived at our selection of 34 edits organically. From very early on, `hs-to-coq` had support for some form of edits. Initially, this was limited to *renamings*, which simply specified types or values that ought to be renamed: `type HsType = coq_type` or `value hsValue = coq_value`. However, the need for other edits quickly became apparent, with the early version of the edit language containing only two edits, both of which have since been deprecated and are now unused: `type synonym`, which specifies the result kind of a type synonym (Section 8.4.5); and `data type arguments`, which allows the user to customize the parameters and indices of a data type definition (Section 8.7.7). You may notice this omits renamings; at the time, renamings were a separate category of input, and were stored in a separate file.

Obviously, things have changed.

From the start, edits were intended to ease the process of verification. With the initial target of `hs-to-coq` being GHC, we needed the ability to restrict the scope of the verification. As we extended the target of our verification – adding examples, pivoting to the `base` library, moving on to `containers`, and turning back to GHC – we regularly found that we needed more and more powerful edits. Our edit language – and `hs-to-coq` itself – coevolved with our verification projects, growing more powerful each time we encountered stumbling blocks. During the course of this evolution, we often took advantage of the breadth of our collaboration: I would work on improving `hs-to-coq` and the edit language as my collaborators pushed them to new heights.

Eventually, we reached a point where new edits became "optional"; for instance, consider the `set type` edit. This edit allows the user to change the type of a translated definition, but nothing else about it. For a period, we wanted this edit but did not have it; however, we could work around this lack with `redefine`, meaning that `set type` became lower priority. This indicates two things: first, that at some point the edit language became "sufficiently expressive" to talk about almost everything we wanted to do; and second, that "sufficiently expressive" wasn't the only target we had in mind.

Eventually, what seemed to drive a need for new edits was when we found ourselves wanting to make a *systematic* change to the code base. By adding edits

such as `redefine` and `rewrite`, we could make one-off code changes where necessary, so we no longer needed to update `hs-to-coq` every time our users (ourselves) needed to make a new code change. But repetition is the bane of the programmer in general, and more so when considering our desire for a mechanized formalization gap. The benefit of a mechanized formalization gap is that it allows for examination of the edit files to understand what is happening; writing `in Mod.value1 rewrite Lib.hsName = coq_name` for a dozen different `Mod.value`s is much less clear than writing `rename value Lib.hsName = coq_name` once, and has a greater danger of missing a case. It also provides greater confidence, and greater resilience to code changes: `redefine` replaces an entire definition, and thus would not pick up optimizations that might need to be verified, whereas `set type` would leave those changes alone.

It is also important to remember that edits are not the only code transformations that `hs-to-coq` applies. Some transformations are carried out automatically: `Require`-ing modules; placing type-level definitions before value-level definitions; reordering definitions in dependency order; renaming reserved words to add an underscore; implementing pattern matching with guards and fallthrough in terms of local functions; transforming type classes into a CPSed representation so that they compute; and other such universally-present details. We also provide two more channels for input: preambles and midambles. These are Coq files, conventionally called `preamble.v` and `midamble.v`, whose contents are pasted into the generated code. The difference between the two is that the preamble comes before all the translated code but after module imports, and the midamble comes in between the translated types and the translated values. These are like the `add type` and `add` edits, but they are not parsed and do not participate in dependency calculation. We often subsume these features under the umbrella of the edit language when discussing `hs-to-coq`, but they are meaningfully distinct.

In the end, our edit language – and these other features, but the edit language in particular – is powerful and useful. It can capture most of the code changes we want to make in practice, enabling us to verify code we would otherwise have no hope of getting started with. But the edit language is clearly evolved, and future work might benefit from a rethinking of the design from scratch. There are distinctions that we draw informally which could be reflected in the language; for example, we informally distinguish between edits that affect only the implementation and edits that affect the interface. The former, such as `rename` applied to a local variable, do not need to be seen by other modules; the latter, such as `rename` applied to an exported variable, do. That these are mixed together is unfortunate. We might want to think about making the design of the edits more orthogonal; for example, our `redefine` edit is the same as `skip` plus `add` or `add type`. We could instead think about structuring the edits in a systematic way, so that finding the edits necessary to address a problem was less of an ad hoc process. In general, the edit language that we have could serve as the core of a future cleaner redesign thereof.

4.3. The general form of edits

To read edit files, as for any programming language, we need to understand their syntax. An edit file consists of a sequence of individual *edits*, one per line. For example, sampling from the global edits we apply everywhere (which are found in the file `examples/base-src/edits` in the `hs-to-coq` repository):

```
skip module GHC.Magic

skip class Foreign.Storable.Storable

rename type  GHC.Tuple.() = unit
rename value GHC.Tuple.() = tt
```

These four edits cause `hs-to-coq` to (1) refrain from translating the module `GHC.Magic`, which contains only low-level functions; (2) omit the definition of the `Storable` type class, which is used for marshalling values to and from raw bits, along with all of its instances; (3) replace the Haskell `()` type with the Coq `unit` type; and (4) replace the Haskell `()` constructor with the Coq `tt` constructor.

As a rule, edits are exactly one line long, ending at the newline. In this dissertation, I am constrained by the page width, so I sometimes add line breaks to edits; I denote these with ↩ (as briefly mentioned in Section 2.3). The exception to this one-edit-one-line rule is that certain edits involve writing Coq code directly in an edit file; these Coq definitions are allowed to span more than one line.

4.4. The semantics of edits

Because each edit makes its own distinct changes, understanding the semantics of edits depends largely on understanding the individual edits in question, which we discuss later in this chapter. Nevertheless, some general principles hold true. Each edit is a systematic operation on a codebase; it may have a global or a local effect, but what it does is consistent. With only two exceptions, each edit operates on the *translated* code; this matters, for example, when referring to argument names (`arg_0__`, etc., before pattern matching) or operators (we must refer to `op_zpzp__` instead of `++`). Additionally, names must always be fully qualified in order to match; `skip op_zpzp__` will not do anything, but `skip GHC.List.op_zpzp__` will omit the `(++)` operator. At least, it will if this edit is included while translating `GHC.List`: the final piece of the semantics of edits is that they apply only if included during translation. While `hs-to-coq` retains some information in `.h2ci` files so that it can be used by later modules, the exact edits applied are not saved. This is deliberate: certain edits might only apply in the body of the module, such as using `rename` to configure the meaning of `Integer` locally, while others might be broadly applicable. Thus, we require the user to differentiate these and include broadly-applicable edits in later modules as appropriate. We have found this local edit/global edit distinction to be valuable, and it would be good to make it more semantically relevant within `hs-to-coq` in future work.

4.5. Using edits

The edit language of `hs-to-coq` is rich, and presenting its power and flexibility is one of the chief goals of this thesis. We saw in Section 1.1 and Chapter 2 how to use edits to customize verification in simple examples; in the next three chapters, we will see how `hs-to-coq` and its edit language grew over time to be able to apply to complex real-world code. At the same time, these three chapters will be an *evaluation* of the edit language. The goal of the edit language is to enable the translation (and then verification) of realistic Haskell programs; the best way to see how well we met this goal is to see `hs-to-coq` and its edit language in action. And the reason that `hs-to-coq` does meet this goal is the other piece highlighted in these chapters: the edit language is not arbitrary, but has co-evolved with the problems we were attempting to solve. This co-evolution ensured that the edit language was useful every step of the way; in order to see if we have avoided over-fitting the edit language to specific problems, we also look back at each step to see what we needed to add to reach our new goals.

Once we have seen all these ways we successfully used the edit language, I will present an explanation of the behavior of each of the 34 different edits in Chapter 8; for each edit, the explanation includes the syntax of the edit, an explanation of what it does, and concrete examples of how it behaves when applied to specific pieces of code.

CHAPTER 5

"Total Haskell is Reasonable Coq"

In these next three chapters, we explore the ways `hs-to-coq` has evolved over time. This evolution involved a change in both our methodology and our targets – in both *how* we used `hs-to-coq` and *what* we used it on. In each of these chapters, we will focus on a different period in `hs-to-coq`'s development. First, in this chapter, we focus on pure, total code, as seen through the `base` library and several smaller examples. In Chapter 6, `hs-to-coq` graduates to working on the `containers` library, and, forced to contend with realistic code, considers the question of total interfaces implemented through impure code. Finally, in Chapter 7, `hs-to-coq` development attains its modern form: enthusiastic use of edits and a willingness to address fundamentally partial code.

This trajectory through history also functions as a trajectory from simple to complex. The further we move from pure and total code, the more demands we place on `hs-to-coq`, and the more features we use; indeed, we added these features to support these applications. As we go through these chapters, we will see how to apply the `hs-to-coq` and its edit language to progressively more complex pieces of software, demonstrating more and more of what it's capable of.

Thus, we now focus on where we began our work with `hs-to-coq`: focusing on its application to pure, total code. In "Total Haskell is reasonable Coq" (Spector-Zabusky et al., 2018), we presented `hs-to-coq` publicly for the first time, and we used it to verify three simple examples. We also explored the design and implementation of `hs-to-coq`, as I have done in Chapter 3, and discuss the translation of the `base` library.

Our initial plan had been to focus on pure and total code because this is what could be expressed natively in Coq. We felt, from personal experience, that a lot of Haskell code was structurally recursive if it was recursive at all, giving us confidence that this would be useful in practice. Since all Haskell code is pure (modulo functions like `unsafeCoerce`), we knew this would not be an obstacle to translation; good Haskell code minimizes the use of `IO`, so we felt there would also be a lot of code that was pure in the no-use-of-`IO` sense as well.

In the end, our results were mixed: yes, a lot of code is structurally recursive and pure. But in practice, changing Haskell into Coq in an unmodified or minimally-modified way simply doesn't go far enough. What we saw when working on "Total Haskell is reasonable Coq" (Spector-Zabusky et al., 2018) was that this vision was fundamentally incomplete – it was not that it didn't work for some code, but that so much code that we wanted to be able to translate and reason about was broader than that. And as we continued to work on `hs-to-coq` (Chapters 6 and 7), we branched out into such code.

But to begin with, we started small, by presenting three examples:

- Reasoning about *type class laws*, by specifying various type classes and proving the `Functor`, `Applicative`, and `Monad` laws for a simple type (Section 5.1).
- Verifying *Hutton's razor*, a simple interpreter and compiler (Section 5.2).
- Verifying *GHC's implementation of bags*, also known as multisets (Section 5.3). (We also saw this example back in Chapter 2, where it served as an introduction to the experience of using `hs-to-coq`; in this chapter, we use that as a jumping-off point and go into more detail.)

5.1. Type class laws

One of the first places a Haskell programmer might come across the notion of proving a theorem about a computer program is around type classes. In particular, the type classes `Functor`, `Applicative`, and `Monad` play a central role in Haskell programming. These type classes come from category theory, and they feature generally-useful methods. For example, the `Functor` type class is (The Core Libraries Committee, 2018, the `Prelude` module; reformatted)

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

  -- | Replace all locations in the input with the same value.
  -- The default definition is @'fmap' . 'const'@, but this may be
  -- overridden with a more efficient version.
  (<$) :: a -> f b -> f a
  (<$) =  fmap . const
```

This type class comes with three different laws:

(1) For all types `a`, we have `fmap (id @a)` ≡ `id`. (The `@a` denotes type application.)
(2) For all functions `f :: a -> b` and `g :: b -> c`, we have `fmap (g . f)` ≡ `fmap g . fmap f`.
(3) For all types `a` and `b`, we have `(<$)` ≡ `fmap . const @a @b`; that is, any specialized implementation of `(<$)` must be equivalent to the default one.

The first two laws, in particular, are part of the definition of a functor in category theory (specifically, an endofunctor on **Hask**, the idealized category of Haskell types and functions).

When a user first encounters `Functor`, they may think that any implementation of `fmap` that type checks is "good enough", but the documentation (The Core Libraries Committee, 2018, the `Prelude` module) quickly clarifies that we need to satisfy the first two laws above. This means that a correct instance of `Functor` requires more than just a function definition: it requires a *proof*.

Well, `hs-to-coq` is in the business of enabling you to provide just that, so it's no surprise that one of our initial applications was on verifying the type class laws for instances thereof. We defined law-carrying subclasses of ten different Haskell type classes, which we divide into four categories:

Structural: `Eq` and `Ord`;
Higher-order structural: `Eq1` and `Ord1`;
Algebraic: `Semigroup` and `Monoid`; and
Category-theoretic: `Functor`, `Applicative`, and `Monad`.

We also include two special-purpose law-like type classes:

- We include a version of the laws for `Eq` that relates them to Coq equality; and
- We include a type class for type constructors that are both `Foldable` and `Functor`s, relating the two.

5.1.1. Lawful type class examples. The definitions of the law-carrying type classes for `Eq` and `Functor` are as follows (up to reformatting):

```
Class EqLaws (t : Type) `{Eq_ t} :=
  { Eq_refl  : reflexive  _==_;
    Eq_sym   : symmetric  _==_;
    Eq_trans : transitive _==_;
    Eq_inv   : forall x y : t, x == y = ~~ (x /= y)
  }.

Class FunctorLaws (t : Type -> Type) `{Functor t} :=
  {
    functor_identity    : forall a (x: t a), fmap id x = x;
    functor_composition :
      forall a b c (f : a -> b) (g : b -> c) (x : t a),
        fmap g (fmap f x) = fmap (g ∘ f) x
  }.
```

The first thing to notice is that we have separated these law-bearing type classes from the operation-bearing Haskell ones. Why? For `hs-to-coq`, this is an easy choice: we want to leave the translated code alone, which means not injecting proofs into every `instance` of `Eq` we find. But in general, this is a standard API design approach in dependently typed languages, corresponding to the distinction between extrinsic and intrinsic verification; it is easier to be able to define a value and then carry out the proofs separately.

We focus on the `EqLaws` class first. Note that we rename the Haskell `Eq` type class to `Eq_` in Coq, because `Eq` is a constructor of the `comparison` data type.[11] We thus see that this type class is a subclass of `Eq_`, and it requires that (a) `(==)` is an equivalence relation (we use the notation `_==_` instead of `(==)` in Coq), and (b) `(==)` and `(/=)` are opposites (the `~~` operator is SSReflect's notation for boolean negation). The functions `reflexive`, `symmetric`, and `transitive` come from SSReflect, and encode the expected properties for computable relations, i.e., for functions of type `A -> A -> bool` for some `A`.

[11] This is a data type we actually use; it corresponds to Haskell's `Ordering` type, except its three constructors are `Lt`, `Eq`, and `Gt` instead of `LT`, `EQ`, and `GT`.

The `FunctorLaws` class was written in a different style, but also clearly directly encodes the two laws for `fmap` we expressed above; it omits the law for `<$`, presumably because we haven't needed it yet in any of our developments. We choose to use Leibniz (propositional) equality, rather than Haskell's boolean-valued `(==)` or any sort of setoid representation, because it is the simplest option, incurring no type class constraints and creating no extra parameters; so far, it has sufficed for our purposes.

5.1.2. Design considerations for law-bearing type classes. The rest of the law-bearing type classes look much the same, but there were some design considerations. First, consider the `SemigroupLaws` type class. Note that we renamed the binary operator for `Semigroup` from `(<>)`, which is propositional inequality in Coq, to `(<<>>)`.

```
Class SemigroupLaws (t : Type) `{ Semigroup t } `{ EqLaws t } :=
  { semigroup_assoc    : forall (x y z : t),
      ((x <<>> (y <<>> z)) == ((x <<>> y) <<>> z)) = true;
  }.
```

As we can see, this type class requires `EqLaws`, so that the associativity law can be expressed up to Haskell-internal equality. This is generally considered to be the semantics of `Semigroup` when available, but such an instance isn't always available: `Semigroup` isn't a subclass of `Eq`, and there can be and are instances of `Semigroup` that both don't and *can't* have an `Eq` instance. Indeed, the following two instances are both widely used:

- `newtype Endo a = Endo { appEndo :: a -> a }`, the type of endomorphisms on the type `a` (i.e., functions from the type `a` to itself), and the corresponding `instance Semigroup (Endo a)`, where the binary operation is function composition.
- `instance Ord a => Semigroup (Set a)`, for the `Set` data type from the `containers` library (see Chapter 6), where the binary operation is set union.

The first `Semigroup` instance *cannot* be proven to be lawful with respect to `Eq`, as in the definition of `SemigroupLaws`, because functions do not have an `Eq` instance as they lack decidable equality. Conversely, the second instance *cannot* be proven to be lawful with respect to propositional equality, as `Set`s are explicitly an abstract data type with observably equal values that have different structures.

For our purposes, we wanted to prove that `Set`, `IntSet`, and `Map` were all lawful instances of `Semigroup` (although there were complications (Breitner et al., 2018)), so we went with the version you see, which respects `Eq`; however, this was not the only possible design choice.

Now, consider `MonadLaws`, recalling that `Monad` is a subclass of `Applicative`, and so transitively of `Functor`.

```
Class MonadLaws (t : Type -> Type)
                `{!Functor t, !Applicative t, !Monad t,
                  !FunctorLaws t, !ApplicativeLaws t} :=
  { monad_left_id :
      forall A B (a :A) (k : A -> t B),
```

```
      (return_ a >>= k)  =  (k a);
  monad_right_id :
    forall A (m : t A),
      (m >>= return_)  =  m;
  monad_composition :
    forall A B C (m : t A) (k : A -> t B) (h : B -> t C),
      (m >>= (fun x => k x >>= h)) =  ((m >>= k) >>= h);
  monad_applicative_pure :
    forall A (x:A),
      pure x = return_ x;
  monad_applicative_ap :
    forall A B (f : t (A -> B)) (x: t A),
      (f <*> x) = ap f x
}.
```

The first thing we see is that we require that any instance of `MonadLaws` also be an instance of `FunctorLaws` and `ApplicativeLaws`. This is unsurprising: a lawful monad is a lawful applicative which is a lawful functor, and so we can always provide these instances. This is not the only possible design constraint – we could decouple `FunctorLaws`, `ApplicativeLaws`, and `MonadLaws` from each other. However, this would likely be a bad idea! As the Haskell world learned over many years, when type classes that are guaranteed to be related are instead defined to be disjoint, the result is inevitably headaches and code duplication. This is why the Functor-Applicative-Monad proposal was enacted in 2014, making `Functor` and `Applicative` superclasses of `Monad`: unifying this type class hierarchy became necessary to avoid these problems (HaskellWiki contributors, 2015). We do not need to recapitulate our own language's design mistakes.

One other part of the superclass constraints that stands out are the `!`s; these shut down extra type class inference, thereby preventing the diamond problem. Specifically, note that `Applicative t` requires an implicit `Functor t` member, as do `Monad t`, `FunctorLaws t`, and `ApplicativeLaws t`. By default, these would each be generalized as implicit parameters; using the `!` suppresses that, and so the given instance is used.

Now, we can look at the laws. The first three laws presented are indeed the monad laws for unit/`return_` (renamed because `return` is a keyword in Coq) and bind/`(>>=)`, but there are two more laws: `monad_applicative_pure` and `monad_applicative_ap`, which link `Monad` to its superclass. This is required (and in the documentation) to ensure that we can translate between using the methods of `Monad` and `Applicative` freely. (We can also translate to and from using the methods of `Functor`, as `ApplicativeLaws` contains similar methods.)

Note also that these laws are phrased in terms of propositional equality. This is the opposite design choice from that we made for `SemigroupLaws`, even though the same concerns apply. We make this choice because *so many* monad instances contain functions; we would be unable to work even with common monads like `Reader` and `State` if we used `==` for equality testing. However, because we want to equate

functions, we in practice require functional extensionality in our development. This corresponds to how Haskell programmers think about functions, so it is not a terrible requirement, but it is something worth highlighting.

Finally, there are two type classes whose absence is notable: `Alternative` and `MonadPlus`. These type classes would seem to be natural fits for the same paradigm we have just seen. However, `Alternative` provides four methods, not just two (The Core Libraries Committee, 2018, the `Control.Applicative` module):

```
class Applicative f => Alternative f where
    -- | The identity of '<|>'
    empty :: f a
    -- | An associative binary operation
    (<|>) :: f a -> f a -> f a

    -- | One or more.
    some :: f a -> f [a]
    some v = some_v
      where
        many_v = some_v <|> pure []
        some_v = liftA2 (:) v many_v

    -- | Zero or more.
    many :: f a -> f [a]
    many v = many_v
      where
        many_v = some_v <|> pure []
        some_v = liftA2 (:) v many_v
```

The first two methods, `empty` and `(<|>)`, are the methods we think of when we think of `Alternative`. However, the last two, `some` and `many`, are mutually coinductive and produce infinite lists. This is something `hs-to-coq` cannot handle, so we decided to skip `Alternative` entirely. Another option would be to skip just the methods `some` and `many`, and skip all functions that use them. Skipping `Alternative` as we did means we also skipped `MonadPlus`, as the latter is a subclass of the former.

5.1.3. Unusual law-bearing type classes. Lastly, we consider the two law-bearing type classes that fall outside the `Laws` scheme. First is the type class that relates Haskell's decidable (boolean-valued) equality, `(==)`, to Coq's propositional Leibniz equality (=), which is named `EqExact`:

```
Class EqExact (t : Type) `{EqLaws t} :=
  { Eq_eq : forall x y : t, reflect (x = y) (x == y) }.
```

Again, this type class was written using SSReflect. The `reflect` type takes as arguments a `Prop`osition and a `bool`ean, and says that the proposition holds if and only if the boolean is true. If we have an instance of this type class, we can switch

freely between "true" equality and Haskell equality; this makes life much easier, but of course is not always available.

Finally, we look at `FoldableFunctor`, the type class that relates `Foldable` and `Functor`. The `Foldable` type class represents type constructors that can be "folded together"; this can be given solely through providing an implementation of `foldr` or `foldMap` for the type (although there are many other methods that can be defined for efficiency's sake). The more familiar `foldr` has the type

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

whereas `foldMap` has the type

```
foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m
```

Both express that values of type `t a` contain a number of `a`s that can be combined, sequentially, into a single one.

(In)famously, `Foldable` doesn't come with any laws of its own, but the documentation requires that *if* a type constructor is an instance of both `Foldable` and `Functor`,[12] then "it should satisfy `foldMap f = fold . fmap f`" (The Core Libraries Committee, 2018, the `Foldable` module), where the method

```
fold :: (Foldable t, Monoid m) => t m -> m
```

folds a data structure that already contains values from a monoid. We express this in the `FoldableFunctor` type class:

```
Class FoldableFunctor {t} `{Foldable t} `{Functor t} := {
  foldfmap : forall a b (f : a -> b)`{Monoid b},
               foldMap f = fold ∘ fmap f;
}.
```

5.1.4. Successors. In "Total Haskell is reasonable Coq" (Spector-Zabusky et al., 2018), we focused on presenting the `Functor`, `Applicative`, and `Monad` laws for one simple type in particular: the `Succs` type, from the `successors` library on Hackage (Breitner, 2017). The verification of this type was done by my coauthor, Joachim Breitner, who also authored the library.

The `Succs` type represents the "successors" of an element with respect to a nondeterministic relation (Breitner, 2017, line 27 in module `Control.Applicative.Successors`):

```
data Succs a = Succs a [a] deriving (Show, Eq)
```

The library pulls out these two components with the accessors `getCurrent` and `getSuccs`, so we call the `a` field of the constructor the "current" element, and the elements of the list the "successors".

As an example, suppose we relate all letters from `'A'` to `'Z'`, in both upper and lower case, with the next letter of either case. Then we would have

[12] Why isn't `Foldable` a subclass of `Functor`? Because some types are instances of the former but not the latter. Again, consider `Set` from `containers`. We can consume its values without incurring an `Ord` constraint, so we can provide a `Foldable` instance. However, `fmap` would need to produce a new `Set b` for *any* `b`, which is impossible as we don't have an `Ord` instance available.

```
succeed :: Char -> Succs Char
succeed c
  | 'a' <= c && c < 'z' || 'A' <= c && c < 'Z'
    = Succs c [toUpper $ succ c, toLower $ succ c]
  | otherwise
    = Succs c []
```

so that `succeed 'a' == Succs 'a' "Bb"`, `succeed 'B' == Succs 'B' "Cc"`, and so on, culminating in `succeed 'Z' == Succs 'Z' ""`.

The definition of `Functor` for `Succs` is trivial, but what does the `Applicative` instance do? Remembering that the idea behind `Succs` is to model *one step* in a relation, `Succs f fs <*> Succs x xs` should be "at" `f x`, but then *either* `x` should be passed to the `fs` *or* `f` should be applied to `xs`. This way, *only one* step is taken. Put another way, we should have:

```
Succs f [f'1, f'2, ..., f'M] <*> Succs x [x'1, x'2, ..., x'N]
 = Succs (f x) [ f'1 x,   f'2 x,   ..., f'M x
               , f   x'1, f   x'2, ..., f   x'N ]
```

This is implemented by the instance (Breitner, 2017, lines 32–34 in the module `Control.Applicative.Successors`; reformatted)

```
instance Applicative Succs where
  pure x = Succs x []
  Succs f fs <*> Succs x xs =
    Succs (f x) (map ($x) fs ++ map f xs)
```

(The definition of `pure` simply corresponds to an element with no successors.)

This is perhaps more clearly phrased in terms of `liftA2 (,)` (which is also an alternative presentation of `Applicative`). The above definitions mean that the expression `liftA2 (,) (Succs x xs) (Succs y ys)` will produce a value whose current location is `(x,y)` and whose children are pairs where *exactly one side* has taken one step:

```
liftA2 (,) (Succs x [x'1, x'2, ..., x'M])
           (Succs y [y'1, y'2, ..., y'N])
  = Succs (x,y) [ (x'1, y),   (x'2, y),   ..., (x'M, y)
                , (x,   y'1), (x,   y'2), ..., (x,   y'N) ]
```

The monad instance is slightly more obtuse, but intuition can come more easily from `join` than bind. If we have a `Succs` containing more `Succs`es, then we have the information about those values *two* steps away from the starting value. This means that when we `join`, we want to take the successors of the current element and the current elements on the successor, but no more – the successors of the successors are *two* steps ahead, not one! In other words, we want:

```
join (Succs (Succs x [x'1, x'2, ..., x'M])
            [Succs y1 _, Succs y2 _, ..., Succs yN _])
  = Succs x [y1, y2, ..., yN, x'1, x'2, ... x'M]
```

Note that we don't even have to name the successors of the successors.

In terms of `(>>=)`, instead suppose we have `Succs x xs >>= f`. Then the current element becomes the current element of `f x`, and the one-step successors are both (a) the current elements of applying `f` to `xs`, the successors of `x`, and (b) the successors of `f x`. In instance form, this is:

```
instance Monad Succs where
    Succs x xs >>= f = Succs y (map (getCurrent . f) xs ++ ys)
      where Succs y ys = f x
```

Again, we never use the successors of `f` applied to the successors, because we want to only advance one step.

The fact that this type with these behaviors forms a lawful `Applicative` and `Monad` instance is, perhaps, not immediately obvious. Breitner (2017) provided 74 (nonblank) lines of equational reasoning proofs in a comment at the bottom of the `Control.Applicative.Successors` module, so we can be confident, but we would of course like to promote this specification from a moribund comment to living Coq.

And so we (including Breitner) did, with `hs-to-coq`. We translated the module `Control.Applicative.Successors` into Coq, and provided instances of `FunctorLaws`, `ApplicativeLaws`, and `MonadLaws` for `Succs`. The translation of this module is *almost* short enough to be presentable, at 130 lines (including blanks and comments) using a modern `hs-to-coq`.[13] At the time, we did not need very many edits; now, we only need two:

```
rename value Control.Applicative.Successors.Succs = ↩
  Control.Applicative.Successors.succs
order Control.Applicative.Successors.Functor__Succs ↩
      Control.Applicative.Successors.Applicative__Succs_liftA2 ↩
      Control.Applicative.Successors.Applicative__Succs ↩
      Control.Applicative.Successors.Monad__Succs_return_
```

The first takes care of the type–constructor pun so common in Haskell code; the latter provides an order between some of the different method definitions and type class instances, as syntactic dependency analysis doesn't work for those implicitly-passed arguments. The type itself translates almost exactly as a Coq programmer would write it (though some would leave out the vertical bar):

```
Inductive Succs a : Type := | succs : a -> list a -> Succs a.
```

The type class instances suffer from our verbose, continuation-style encoding of type classes (Section 3.7), but they too are straightforward. For example, consider `Applicative`:

```
Local Definition Applicative__Succs_op_zlztzg__
    : forall {a} {b}, Succs (a -> b) -> Succs a -> Succs b :=
  fun {a} {b} =>
    fun arg_0__ arg_1__ =>
```

[13] A reasonable threshold might be two pages of code, which is 84 lines.

```
      match arg_0__, arg_1__ with
      | succs f fs, succs x xs =>
          succs (f x)
                (Coq.Init.Datatypes.app
                  (GHC.Base.map (fun arg_2__ => arg_2__ x) fs)
                  (GHC.Base.map f xs))
      end.

Local Definition Applicative__Succs_liftA2
    : forall {a} {b} {c},
        (a -> b -> c) -> Succs a -> Succs b -> Succs c :=
  fun {a} {b} {c} =>
    fun f x => Applicative__Succs_op_zlztzg__ (GHC.Base.fmap f x).

Local Definition Applicative__Succs_op_ztzg__
    : forall {a} {b}, Succs a -> Succs b -> Succs b :=
  fun {a} {b} =>
    fun a1 a2 => Applicative__Succs_op_zlztzg__
                   (GHC.Base.id GHC.Base.<$ a1) a2.

Local Definition Applicative__Succs_pure
    : forall {a}, a -> Succs a :=
  fun {a} => fun x => succs x nil.

Program Instance Applicative__Succs : GHC.Base.Applicative Succs :=
  fun _ k__ =>
    k__ {| GHC.Base.liftA2__ :=
             fun {a} {b} {c} => Applicative__Succs_liftA2 ;
           GHC.Base.op_zlztzg____ :=
             fun {a} {b} => Applicative__Succs_op_zlztzg__ ;
           GHC.Base.op_ztzg____ :=
             fun {a} {b} => Applicative__Succs_op_ztzg__ ;
           GHC.Base.pure__ :=
             fun {a} => Applicative__Succs_pure |}.
```

The odd `op_` names are the translations of the infix operators, as was explained in Section 3.5; the name `op_zlztzg__` corresponds to `(<*>)`, and the name `op_ztzg__` corresponds to `(*>)` (analogous to `(>>)` and possessed of a default definition). Also note that the `Applicative` class contains, in addition to `pure` and `(<*>)`, both the aforementioned `(*>)` and also `liftA2`. Again, both of these have been provided with their default definitions.

The Coq proofs – that is, the instances of the law-bearing type classes – were fairly straightforward, consisting largely of repeated rewriting. Indeed, this was exactly the

point: we were able to, without very much effort, take a real – if simple – Haskell library, examine the proofs that the author had written into its comments, and proceed with mechanical verification. Exactly as we had hoped to use `hs-to-coq`!

5.2. Hutton's Razor

"Hutton's Razor", from *Programming in Haskell* (Hutton, 2016, ch. 16, pp. 241–246), is a simple language equipped with both an interpreter and a compiler. The entire original example is only a handful of lines long; I present it in its entirety, and then explain it.

```
data Expr = Val Int | Add Expr Expr

eval :: Expr -> Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y

type Stack = [Int]

type Code = [Op]

data Op = PUSH Int | ADD
          deriving Show

exec :: Code -> Stack -> Stack
exec [] s = s
exec (PUSH n : c) s = exec c (n : s)
exec (ADD : c) (m : n : s) = exec c (n+m : s)

comp :: Expr -> Code
comp (Val n) = [PUSH n]
comp (Add x y) = (comp x ++ comp y) ++ [ADD]
```

The language itself is given by the type `Expr`, and consists of integer literals (`Val`) and addition expressions (`Add`). We work only with the abstract syntax tree, and not with any sort of source code representation, but we can imagine an AST such as

```
Add (Add (Val 1) (Val 2)) (Add (Val 3) (Val 4))
```

as equivalent to the expression `(1+2)+(3+4)`. The interpreter for this language is given by the `eval` function, and it simply evaluates all the additions in the tree. For the preceding example, this works out to `10`, since `1+2` is `3`, `3+4` is `7`, and `3+7` is `10`.

The next portion of Hutton's razor is a simple stack-based language, used as a compilation target for `Expr`. The execution model for this language is that we have a program, which is a list of stack operations, and a stack of values. Each instruction is executed in turn, and it may modify the top of the stack; when there are no more instructions, the resulting stack is the result of the program.

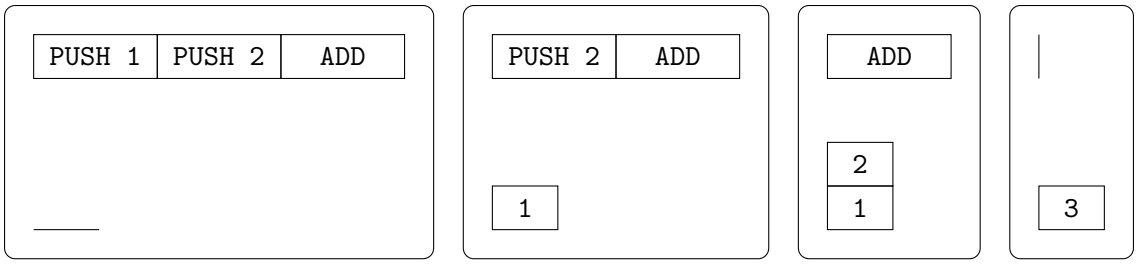

FIGURE 5.1. Evaluation of a stack-based program that adds `1` and `2`. Each state is presented in a rounded rectangle, and is a combination of a program and a stack. The program is presented horizontally at the top of the state, with the operations to be evaluated from left to right. The stack is presented vertically at the bottom of the state, growing upwards. The evaluation of the program proceeds from left to right.

Since a program in this language is a list of stack operations, Hutton needs a type for these operations, which he calls `Op`; the type synonym `Code`, which is a list of `Op`s, represents a stack-machine program. The stack is represented as a list of integers; for clarity, it is referred to with the type synonym `Stack`.

The two stack operations are `PUSH :: Int -> Op`, which takes an integer argument and pushes it onto the stack, and `ADD`, which pops the top two items from the stack and pushes their sum. These meanings are implemented by the function `exec :: Code -> Stack -> Stack`, which takes a program and an initial stack and returns the final stack once every operation in the program has been executed.

Let's consider an example. To represent `1+2`, we say `[PUSH 1, PUSH 2, ADD]`; this is `1 2 +` in reverse Polish notation. Evaluating this program is shown in Figure 5.1. At first, the stack will be `[]`; after evaluating `PUSH 1` it will be `[1]`; and after evaluating `PUSH 2`, it will be `[2,1]`, with the head of the list being the top of the stack. Finally, `ADD` will pop `2` and `1`, sum them to get `3`, and push that result, leaving the stack as `[3]`. Now, recall our example from above: `(1+2)+(3+4)`. The `Code` program that corresponds to this expression would be

```
[PUSH 1, PUSH 2, ADD, PUSH 3, PUSH 4, ADD, ADD]
```

The first three operations place `3` on the stack; the next three operations place `7` on the stack without touching the `3`; and the last operation adds the `3` and the `7`, replacing them with `10`.

Finally, the last function that Hutton presents is `comp :: Expr -> Code`, which compiles the expression-based language to the stack-based language. Consider our example `Expr` term from above, which represents `(1+2)+(3+4)`:

```
Add (Add (Val 1) (Val 2)) (Add (Val 3) (Val 4))
```

The left subtree compiles to `[PUSH 1, PUSH 2, ADD]`, the right subtree compiles to `[PUSH 3, PUSH 4, ADD]`, and so the whole thing compiles to

```
[PUSH 1, PUSH 2, ADD, PUSH 3, PUSH 4, ADD, ADD]
```

And this is exactly the `Code` program we came up with earlier.

5.2.1. Compiler correctness. The goal of Hutton's razor is to prove that `comp` is "correct"; here, that means that `eval` and `exec` have the same result. We have to specify exactly what that means, since `exec` returns a stack of values instead of a single value, and so we say that it should return a singleton stack; `exec` also requires an initial stack, which we leave empty.

THEOREM (Hutton's Razor compiler correctness). *For all* `Expr` *programs* `e`*, the result of* `exec (comp e) []` *is equal to* `[eval e]`*.*

The proof of this theorem follows from a straightforward inductive argument, although it requires strengthening the statement of the theorem to interact with the stack: the result of the compiled program should place the result *on top of* the stack, and not modify what was already there.

THEOREM (Hutton's Razor inductive hypothesis). *For all* `Expr` *programs* `e` *and* `Stacks` `s`*, the result of* `exec (comp e) s` *is equal to* `eval e : s`*.*

PROOF. This proof proceeds by induction on `e`. We consider the two possible constructors of `e`: it is either of the form `Val n`, for some `Int` value `n`, or it is of the form `Add x y`, for some `Expr` programs `x` and `y`.

In the first case, `Val n` evaluates to `n`, and compiles to `[PUSH n]`. Executing `PUSH n` with the stack `s` results in `n : s`, pushing `n` on top of the stack, which is what we wanted.

In the second case, `Add x y` compiles to the concatenation of

(1) The compilation of `x`,
(2) The compilation of `y`, and
(3) `[ADD]`.

It is clear that `exec (p1 ++ p2) s` is the same as `exec p2 (exec p1 s)` – first evaluate all of `p1`, which results in a new stack, and then evaluate all of `p2`. Inductively, this means that the results of compiling `x` and `y` must place the result of `eval x` and `eval y`, respectively, on the stack: `exec (comp x ++ comp y) s` must be equal to `eval y : eval x : s`. Then executing `ADD` on the resulting stack sums these, producing `(eval x + eval y) : s`. Since `eval (Add x y)` will also produce the sum `eval x + eval y`, this is correct. □

This nice, inductive argument should be right at home in Coq, although we would need to prove our statement about the interaction of `expr` and `(++)` more explicitly. However, this was not true when we were first verifying this code, because of an important detail I glossed over above: `exec` is a partial function.

5.2.2. Partiality vs. totality. The definition of `exec` only has three cases: one for an empty program, one for `PUSH`, and one for `ADD`. However, the `ADD` case is the problem: it *assumes* there are two elements on top of the stack for it to pop! This means that evaluating the program `[ADD]` on the empty stack will crash: `exec [ADD] []` crashes with the error "Non-exhaustive patterns in function `exec`". (A fact of which Hutton is aware.) This doesn't harm the correctness theorem because `exec` *is* total on the restricted domain of "outputs of the compiler". Nevertheless, this means that the

straightforward translation of Hutton's Razor into Coq will fail, because Coq cannot handle non-exhaustive pattern matching. What, then, can we do?

Our options now to handle partiality are much more developed, and the translation would in fact succeed both soundly and automatically thanks to the `Default` technology we introduced in our work on `containers` (Chapter 6). Even so, there would still be difficulties, as we will see below.

Regardless, back then, we could not handle this function so directly, and so the question of how to handle and think about this partial version of Hutton's razor became one of the first real conflicts within the `hs-to-coq` team: how should we handle partiality?

By default, at the time, `hs-to-coq` handled partiality in something of a brute-force manner, which we no longer use: when it detected a partial pattern match, it would insert

```
Axiom patternFailure : forall {a}, a.
```

at the top of the file, and then fill in the missing pattern match with `patternFailure`. This means that the translation of `exec` was, at the time[14]

```
Definition exec : Code -> Stack -> Stack :=
  fix exec arg_4__ arg_5__
        := match arg_4__ , arg_5__ with
             | nil , s =>
                 s
             | cons (Mk_PUSH n) c , s =>
                 exec c (cons n s)
             | cons Mk_ADD c , cons m (cons n s) =>
                 exec c (cons (GHC.Num.op_zp__ n m) s)
             | _ , _ =>
                 patternFailure
           end.
```

(A few other differences can be seen, relative to the current version of `hs-to-coq`: we were also prefixing every constructor with `Mk_` to avoid name collisions, something we later stopped doing; we translated infix operators to their desugared prefix name, not to an infix notation; and our numbering for fresh variables was global, rather than function-local, leading to `arg_4__` instead of `arg_0__`.)

Now, this axiom is obviously unsound in the most dramatic way possible. The idea, however, is that because `patternFailure` is used only within the module, and is only generated by `hs-to-coq`, that we know it is being used "responsibly" – that while it might introduce unsoundness (imagine using this translation for `head`), it is "good enough". And importantly, "good enough" is good enough because, at the time, this was supposed to be a *temporary state*: `hs-to-coq` was only responsible

[14]Commit `9c25c13db631a26d3e078fe11a413c8217a2e538`; a line break was added after each match pattern to fit on the page.

for translating total code, and sources of partiality like that were bugs.[15] The point of `patternFailure` was to enable easier development, and so (the thinking went) the unsoundness was not a problem. Once we were done with a project, the final output would not have `patternFailure`, and so would not be unsound; this was only a pragmatic technique to aid in the process of verification.

And this was all true to some extent: it was possible to prove the Coq theorem

```
Theorem comp_correct: forall e,
    exec (comp e) [] = [eval e].
```

with the partial, `patternFailure`-laden version of `exec`. This proves *some* sort of point, but it's not clear what: Does the fact that a proof can go through without relying on `patternFailure` mean that `patternFailure` is harmless, which makes it a good idea for helping the user during development? Or does that mean that `patternFailure` is insidious, which makes it a bad idea because it becomes easy to leave unsoundness lurking in what should be a completed development?

Furthermore, the ability to complete the proof was very sensitive to the exact details of *which* proof we used. The proof by induction presented above does *not* in fact succeed – it gets stuck, because Coq doesn't know what `exec p2 patternFailure` evaluates to when looking at the interaction of `exec` and `++`. Instead, the proof we used is based on the lemma

```
Lemma comp_correct_helper: forall e s d,
    exec (comp e ++ d) s =  exec d (eval e :: s).
```

Instead of declaring this version, with `patternFailure`, the "blessed version", we instead decided to look at the situation as though we were fixing a bug and updated `exec` to be a total function, using `Maybe`:

```
exec :: Code -> Stack -> Maybe Stack
exec [] s = Just s
exec (PUSH n : c) s = exec c (n : s)
exec (ADD    : c) (m : n : s) = exec c (n+m : s)
exec (ADD    : c) _ = Nothing
```

The only changes are the use of `Just` in the `exec [] s` case, and the addition of a final case that returns `Nothing`. With this, all our functions are total, and the proofs are *essentially* unchanged: instead of showing that we avoid a crash, we show that we avoid `Nothing`. However, we do need to change the theorem statements:

Lemma (Hutton's Razor inductive hypothesis, total version). *For all* ***`Expr`*** *programs* `e` *and* ***`Stack`****s* `s`*, the result of* ***`exec (comp e) s`*** *is equal to* ***`Just (eval e : s)`****.*

Theorem (Hutton's Razor compiler correctness, total version). *For all* ***`Expr`*** *programs* `e`*, the result of* ***`exec (comp e) []`*** *is equal to* ***`Just [eval e]`****.*

The distributivity lemma for `exec` and `(++)` must also be modified: we say that `exec (p1 ++ p2) s` is the same as `exec p2 =<< exec p1 s`. This captures exactly

[15] As mentioned, there was already disagreement amongst the team about how to treat partiality at this time, but we were still mostly working with this model for the moment.

the place where Coq failed with `patternFailure`: unlike with an axiom, we know explicitly that `exec p2 =<< Nothing` is `Nothing`.

Even using a modern `hs-to-coq`, however, we would still run into issues. We would, as mentioned above, automatically get a sound translation of `exec`; however, we would still have the issue that we would be unable to prove the lemma `comp_correct_helper` which says that `exec (p1 ++ p2) s = exec p2 (exec p1 s)`. Although the modern definition of `patternFailure` has type `forall {a} `{Default a}, a`, Coq still cannot tell what the value of `exec p2 patternFailure` is. This highlights the difference between using the `Default` type class and having a true bottom value ($\bot$) such as that returned by `error`, as we saw at the end of Section 3.8: here we see the distinction between $\bot$ being a *distinct* inhabitant of every type and `default` being an *unknown but existing* inhabitant of `Default` types with respect to evaluation. While Haskell functions must be continuous with respect to $\bot$, and the result of evaluating them at $\bot$ (which is either a value or $\bot$) is part of their definition, this is not the case with `default`; instead, evaluation that depends on the value of `default` gets stuck, as Coq cannot proceed unless it knows how `default` was constructed.

Hutton's Razor, with its partial `exec` function, is written in an older Haskell style. This use of partial functions with demarcated domains of totality was a common way to write Haskell (along with, for example, eliding most or all type signatures), but is not seen as much anymore. While partial functions still exist, of course, it seems to me that most Haskell programmers would write the `Maybe`-fied version of `exec` if they were writing a simple example these days. Totality has become considered a greater virtue in the modern Haskell community.

5.3. Bags

Even at this early date, we took our first forays into verifying parts of GHC; in particular, we verified GHC's implementation of *bags*, or unordered multisets, which are actually sequences. We saw the bulk of this translation in Chapter 2, where we presented the type, the edits, the translation, and the general verification scheme. Taking this as a base, expand on the results of this process: what we found and what we verified.

5.3.1. A refresher on `Bags`. To reorient ourselves, let us first recall the definition of the `Bag` type, which we first saw in Figure 2.1:

```
data Bag a
  = EmptyBag
  | UnitBag a
  | TwoBags (Bag a) (Bag a) -- INVARIANT: neither branch is empty
  | ListBag [a]             -- INVARIANT: the list is non-empty
    deriving Typeable
```

Although it claims to be a bag, GHC provides functions (such as `bagToList`, `consBag`, and `snocBag`) that rely on the elements being in a particular order; thus, this type is really an implementation of a sequence.

Translating `Bag` to GHC was locally a straightforward affair, only requiring two edits about the representation of numeric types. The most challenging part of the translation was cleanly slicing the `Bag` module out of GHC. While we had to make use of much more powerful versions of this later, when working on verifying actual compiler properties (Chapter 7), this was our first foray into that sort of excision. Thankfully, since the it didn't depend on much else in GHC, slicing `Bag` out of GHC was still fairly straightforward.

Since `Bag` is an abstract data type, it is unsurprising that it comes with invariants, which we see in the comment above; the invariants combine to tell us that `EmptyBag` is the only empty bag, and nothing empty can appear deeply within the type. We translated this invariant to Coq as a fixpoint:

```
Fixpoint well_formed_bag {A} (b : Bag A) : bool :=
  match b with
  | Mk_EmptyBag                => true
  | Mk_UnitBag _               => true
  | Mk_TwoBags Mk_EmptyBag _   => false
  | Mk_TwoBags _ Mk_EmptyBag   => false
  | Mk_TwoBags l r             => well_formed_bag l &&
                                  well_formed_bag r
  | Mk_ListBag []              => false
  | Mk_ListBag (_ :: _)        => true
  end.
```

Once we did this, we then proceeded to verification, which involved proving both the well-formedness and correctness of the various operations in the module. For example, the module contains a function

```
unionBags :: Bag a -> Bag a -> Bag a
```

that computes the union of two bags; since bags are sequences, this is the same as appending them. This function's well-formedness theorem is

```
Theorem unionBags_wf {A} (b1 b2 : Bag A) :
  well_formed_bag b1 -> well_formed_bag b2 ->
  well_formed_bag (unionBags b1 b2).
Proof. case: b1; case: b2 => * //=; intuition. Qed.
```

and its correctness theorem is

```
Theorem unionBags_ok {A} (b1 b2 : Bag A) :
  bagToList (unionBags b1 b2) = bagToList b1 ++ bagToList b2.
Proof.
  by case: b1 => *; case: b2 => *; rewrite -bagToList_TwoBags.
Qed.
```

They demonstrate that `unionBags` preserves well-formedness and corresponds to list append.

So far, this is what we saw in Chapter 2. Now that we have refreshed our memories, we can move forward: what did we learn?

5.3.2. A documentation bug. While all the definitions in the `Bag` module were correct, we found a very minor discrepancy in the documentation. This discrepancy had no practical effect, but was present for over 24 years (Partain, 1996). The issue lay with the `foldBag` function:

```
foldBag
  :: (r -> r -> r) -- Replace TwoBags with this; should be
                   -- associative
  -> (a -> r)      -- Replace UnitBag with this
  -> r             -- Replace EmptyBag with this
  -> Bag a
  -> r
```

This function is designed to reduce a `Bag` to a single value. The first three parameters can be thought of as how to handle each non-`ListBag` constructor, as specified in the comments. As the lack of `ListBag` indicates, however, these are also a standard set of ways to collapse a collection of elements: in `foldBag t u e`, the function `t` associatively combines any two elements, the function `u` maps from the input type to the combinable type, and the value `e` handles the empty-bag case. For example, `foldBag (+) length 0` will compute the sum of the lengths of the members of a `Bag [a]`, simply returning `0` if the bag is empty.

The documentation for this function presents a "[s]tandard definition" of `foldBag`, which is commented out, and then says in a comment before the actual definition that it is a "more tail-recursive definition, exploiting associativity of 't' ". However, the two definitions aren't *quite* the same, even when `t` is associative. The commented-out definition directly operates on the four constructors in catamorphic style:

```
foldBag t u e EmptyBag        = e
foldBag t u e (UnitBag x)     = u x
foldBag t u e (TwoBags b1 b2) = (foldBag t u e b1) `t`
                                (foldBag t u e b2)
foldBag t u e (ListBag xs)    = foldr (t.u) e xs
```

Thanks to the invariants, this means it will only need to refer to `e` in the case of an empty `Bag` or when there are embedded `ListBag` constructors, and will never recursively stumble upon `e` otherwise.

The second definition, however – the one that is actually used – *does* leverage `e` within the bag structure:

```
foldBag _ _ e EmptyBag        = e
foldBag t u e (UnitBag x)     = u x `t` e
foldBag t u e (TwoBags b1 b2) = foldBag t u (foldBag t u e b2) b1
foldBag t u e (ListBag xs)    = foldr (t.u) e xs
```

The `EmptyBag` and `ListBag` cases are the same between the two definitions, so there's no problem there. But the other two cases differ, and impose extra requirements on `t` and `e`. The `TwoBags` case folds the elements of the left and right bags together in the same order, but with different groupings; it is this case that requires `t` to be associative. So far, this matches the documentation. However, the `UnitBag` case has yet another difference: it uses `t` to combine `u x` with `e`, rather than leaving `u x` unmodified. This is necessary, since the value of `e` changes during the recursion; that's exactly how the new definition can be more tail-recursive. However, it imposes the extra requirement that `e` be a *right identity* for `t`: for all `x`, `t x e = x`.

As long as the argument is well-formed, this is the only extra requirement on the arguments to `foldBag`, since `e` could otherwise only be introduced by `EmptyBag` or a `ListBag []`. However, if the input bag can be ill-formed, then `EmptyBag` can show up on the left of a recursive call in the "standard definition", introducing an `e` as the first argument to `t`. In this case, `e` needs to be an ordinary two-sided identity for `t`. Thus, we have a choice in stating the requirements on the arguments for the two definitions to be equal: we can require that the two implementations of `foldBag` are equal when...

(1) `t` is associative with right-identity `e` and they are applied to a well-formed `Bag`; or `foldBag` is a well-formed `Bag`; or
(2) `t` is associative with (two-sided) identity `e`.

In this case, since `Bag`s are an abstract type, the former definition is probably the one that is actually "intended" in some sense; even so, we could imagine the latter easing the proof burden in the middle of a complex theorem.

Now, this minor discrepancy is never tickled by GHC: `foldBag` is only ever called twice, and both these times – as is likely to be the case most of the time – `e` is the identity for `t`. But the discrepancy with respect to the documentation is nevertheless misleading, and was present for over 21 years. While not quite the triumphant bug-finding we might have hoped for, this does demonstrate that, if we look carefully for specifications, we sometimes find that the specifications the developers *believe* hold of their code aren't quite the ones that actually do.

5.3.3. Verified `Bag` operations. As we described in Section 2.4, each operation on `Bag`s can be verified in up to two ways: for well-formedness and for correctness. Since some functions don't return `Bag`s (e.g., `lengthBag`), we can't verify that these preserve well-formedness, but every function we could verify has a specification. At the same time, we could imagine a function which was correct but didn't produce a well-formed bag; for instance, consider the function

```
evilUnion :: Bag a -> Bag a -> Bag a
evilUnion b1 b2 = TwoBags EmptyBag (TwoBags b1 b2)
```

This function produces an ill-formed result, since it contains an internal `EmptyBag`. However, once translated to Coq, we could easily prove

```
Theorem evilUnion_ok {A} (b1 b2 : Bag A) :
  bagToList (evilUnion b1 b2) = bagToList b1 ++ bagToList b2.
```

This is because `bagToList` is defined for all elements of the `Bag` type, not just well-formed ones. Thus, we need to consider the two different kinds of theorems separately. In Section 5.3.1, we repeated the examples of `unionBag_wf` and `unionBag_ok`, which instantiate this paradigm.

Recall that we defined `well_formed_bag` as a fixpoint that produces a `bool`. Focusing on computational functions such as this, instead of noncomputational `Prop`erties, means that we get automatic simplification of well-formedness preconditions, and can sometimes discharge them fully automatically, all for free. This is a hallmark of SSReflect style, as well, so it is more compatible with the other pieces of the proof script, which were written in the same style.

As `Bag` is a simple data structure, none of the proofs are terribly complicated; they were each at most 8 lines long. We verified nineteen operations on bags;[16] I present here a list of all the bag operations that we verified along with their specifications, both in English and in Coq. Each operation is marked with "wf" if it was verified for well-formedness, and "ok" if it was verified for correctness.

`emptyBag` (wf, ok): The polymorphic empty bag. This is defined to be equal to the `EmptyBag` constructor, so our correctness theorems are phrased in those terms:

```
Theorem emptyBag_wf {A} : well_formed_bag (@Mk_EmptyBag A).
Theorem emptyBag_ok {A} : bagToList (@Mk_EmptyBag A) = [].
```

`unitBag` (wf, ok): Create a singleton bag.

```
Theorem unitBag_wf {A} (x : A) :
  well_formed_bag (unitBag x).
Theorem unitBag_ok {A} (x : A) :
  bagToList (unitBag x) = [ x ].
```

`lengthBag` (ok): Compute the number of elements of a bag.

```
Theorem lengthBag_ok {A} (b : Bag A) :
  lengthBag b = Zlength (bagToList b).
```

`elemBag` (ok): Test if a value occurs in the bag at least once, using the `Eq_` instance for that type. We also provide a stronger theorem, `elemBag_eq_ok`, which applies when the elements of the bag satisfy the `EqExact` type class; this type class says that the Haskell `(==)` method corresponds to true Coq equality. This theorem says that `elemBag` corresponds to the `In` predicate for lists.

```
Theorem elemBag_ok
    {A} `{GHC.Base.Eq_ A} (x : A) (b : Bag A) :
  elemBag x b = any (GHC.Base.op_zeze__ x) (bagToList b).
Corollary elemBag_eq_ok
    {A} `{EqExact A} (x : A) (b : Bag A) :
  elemBag x b <-> In x (bagToList b).
```

[16] Not including `all_bag`, for reasons we discuss later.

unionManyBags (wf, ok): Given a list of bags, takes the union (or concatenation, since Bags are ordered) of all of them.

```
Theorem unionManyBags_wf {A} (bs : list (Bag A)) :
  all well_formed_bag bs ->
  well_formed_bag (unionManyBags bs).
Theorem unionManyBags_ok {A} (bs : list (Bag A)) :
  bagToList (unionManyBags bs) = concat (map bagToList bs).
```

unionBags (wf, ok): Take the union of two bags (which is their concatenation, since Bags are ordered).

```
Theorem unionBags_wf {A} (b1 b2 : Bag A) :
  well_formed_bag b1 -> well_formed_bag b2 ->
  well_formed_bag (unionBags b1 b2).
Theorem unionBags_ok {A} (b1 b2 : Bag A) :
  bagToList (unionBags b1 b2) = bagToList b1 ++ bagToList b2.
```

consBag (wf, ok): Prepend an element to a bag. That both this and snocBag are provided is one of the reasons that we know that Bags are ordered.

```
Theorem consBag_wf {A} (x : A) (b : Bag A) :
  well_formed_bag b ->
  well_formed_bag (consBag x b).
Theorem consBag_ok {A} (x : A) (b : Bag A) :
  bagToList (consBag x b) = x :: bagToList b.
```

snocBag (wf, ok): Append an element to a bag. That both this and consBag are provided is one of the reasons that we know that Bags are ordered.

```
Theorem snocBag_wf {A} (b : Bag A) (x : A) :
  well_formed_bag b ->
  well_formed_bag (snocBag b x).
Theorem snocBag_ok {A} (b : Bag A) (x : A) :
  bagToList (snocBag b x) = bagToList b ++ [x].
```

isEmptyBag (ok): Test if a bag has no elements.

```
Theorem isEmptyBag_ok {A} (b : Bag A) :
  well_formed_bag b ->
  isEmptyBag b = null (bagToList b).
```

filterBag (wf, ok): Compute the subbag containing all elements satisfying a given predicate.

```
Theorem filterBag_wf {A} (p : A -> bool) (b : Bag A) :
  well_formed_bag (filterBag p b).
Theorem filterBag_ok {A} (p : A -> bool) (b : Bag A) :
  bagToList (filterBag p b) = filter p (bagToList b).
```

`all_bag` (ok): Test that every element of the bag satisfies this predicate. We wrote this function by hand and tested it in our original work, which used GHC 8.0.2; however, an `allBag` function was added in GHC 8.2, and I noted at the time that this should be replaced with it once we updated to 8.2. Now that we have, we could update our proofs.

```
Theorem all_bag_ok {A} (p : A -> bool) (b : Bag A) :
  all_bag p b = all p (bagToList b).
```

`anyBag` (ok): Test if any element of the bag satisfies the given predicate.

```
Theorem anyBag_ok {A} (p : A -> bool) (b : Bag A) :
  anyBag p b = any p (bagToList b).
```

`concatBag` (wf, ok): Given a bag of bags, compute their union (or concatenation, since `Bag`s are ordered). Similar to `unionManyBags`.

```
Theorem concatBag_wf {A} (bb : Bag (Bag A)) :
  all_bag well_formed_bag bb ->
  well_formed_bag (concatBag bb).
Theorem concatBag_ok {A} (bb : Bag (Bag A)) :
  bagToList (concatBag bb) =
  concat (map bagToList (bagToList bb)).
```

`catBagMaybes` (wf, ok): Given a bag of `Maybe` (in Haskell) or `option` (in Coq) values, drop all the `Nothing`s/`None`s and unwrap the `Just`s/`Some`s.

```
Theorem catBagMaybes_wf {A} (b : Bag (option A)) :
  well_formed_bag (catBagMaybes b).
Theorem catBagMaybes_ok {A} (b : Bag (option A)) :
  bagToList (catBagMaybes b) =
  flat_map
    (fun o => match o with Some x => [x] | None => [] end)
    (bagToList b).
```

`partitionBag` (wf, ok): The expression `partitionBag p b` is the same as the pair `(filterBag p b, filterBag (not . p) b)` of the positive and negative filters. This function splits the bag `b` into two disjoint subsets, the former in which all elements satisfy the predicate `p` and the latter in which none do.

```
Theorem partitionBag_wf {A} (p : A -> bool) (b : Bag A) :
  let: (bt, bf) := partitionBag p b
  in well_formed_bag bt && well_formed_bag bf.
Theorem partitionBag_ok {A} (p : A -> bool) (b : Bag A) :
  let: (bt, bf) := partitionBag p b
  in (bagToList bt, bagToList bf) =
     partition p (bagToList b).
```

`foldBag` (ok): Reduce a bag to a single element via an associative binary operation. This function was also the target of verification with respect to the "standard implementation" that we saw in Section 5.3.2.

```
Theorem foldBag_ok
    {A R}
    (f : R -> R -> R) (u : A -> R) (z : R) (b : Bag A) :
  associative f ->
  foldBag f u z b = fold_right f z (map u (bagToList b)).
```

`foldrBag` (wf, ok): Reduce a bag to a single element, just like `foldr` (Haskell)/`fold_right` (Coq) for lists. We also proved two well-formedness properties for `foldrBag` which said that if (1) we are folding into a `Bag B` for some `B`, (2) the reducing function preserves well-formedness of `Bag`s, and (3) the base value is a well-formed `Bag`, then the result is a well-formed `Bag`. These lemmas were not strictly necessary, as these properties follow from the correctness of `foldrBag` (and this is why we did not prove them for `foldBag` or `foldlBag`); however, since `foldrBag` was used to implement other functions on `Bag`s, they were useful to have. For the same reason, we proved similar well-formedness theorems for the `foldr` function on lists.

```
Lemma foldrBag_wf
    {A B}
    (p : A -> bool) (f : A -> Bag B -> Bag B) (z : Bag B) :
  (forall x b,
    p x -> well_formed_bag b -> well_formed_bag (f x b)) ->
  well_formed_bag z ->
  forall b : Bag A,
    all_bag p b -> well_formed_bag (foldrBag f z b).
Lemma foldrBag_wf'
    {A B} (f : A -> Bag B -> Bag B) (z : Bag B) :
  (forall x b,
    well_formed_bag b -> well_formed_bag (f x b)) ->
  well_formed_bag z ->
  forall b : Bag A, well_formed_bag (foldrBag f z b).
Theorem foldrBag_ok
    {A R} (f : A -> R -> R) (z : R) (b : Bag A) :
  foldrBag f z b = fold_right f z (bagToList b).
```

`foldlBag` (ok): Reduce a bag to a single element, just like `foldl` (Haskell)/`fold_left` (Coq) for lists.

```
Theorem foldlBag_ok
    {A R} (f : R -> A -> R) (z : R) (b : Bag A) :
  foldlBag f z b = fold_left f (bagToList b) z.
```

`mapBag` (wf, ok): Apply a function to every element of a `Bag`.

```
Theorem mapBag_wf {A B} (f : A -> B) (b : Bag A) :
  well_formed_bag b ->
  well_formed_bag (mapBag f b).
Theorem mapBag_ok {A B} (f : A -> B) (b : Bag A) :
  bagToList (mapBag f b) = map f (bagToList b).
```

`listToBag` (wf, ok): Convert a list into an equivalent `Bag`. This function simply wraps the list in the `ListBag` constructor if the list is nonempty.

```
Theorem listToBag_wf {A} (l : list A) :
  well_formed_bag (listToBag l).
Theorem bagToList_listToBag {A} (l : list A) :
  bagToList (listToBag l) = l.
```

5.3.4. Unverified `Bag` operations. This list covers all the well-formedness (`_wf`) and correctness (`_ok`) theorems that I proved. Some functions, such as `foldBag`, consume `Bag`s but do not produce them, and so did not have well-formedness theorems; even so, every function in that list which could have had a well-formedness theorem proved about it did. However, as is often the case, there are some functions provided in the `Bag` module that remained outside the scope of this verification. Some of these functions are type class methods; for instance, as discussed in Section 2.3, we skip the instance of the `Outputable` type class for `Bag` since we do not verify pretty-printing. We also elided the translation of the `Data` class, which is used for generic programming, as we do by default.

Some other functions we did not verify are the 10 monadic operations on bags: `filterBagM`, `anyBagM`, `foldrBagM`, `foldlBagM`, `mapBagM`, `mapBagM_`, `flatMapBagM`, `flatMapPairM`, `mapAndUnzipBagM`, and `mapAccumBagLM`. As is the custom in Haskell libraries, these functions are mostly monadic variants on the matching `M`-less functions we saw in the previous section; for instance, `mapBagM_` is like `mapBagM` but doesn't return the result, and `mapAndUnzipBagM` maps a monadic function returning pairs over a bag and splits the result into a pair of bags instead of a bag of pairs.

Nothing fundamental prevents us from verifying these monadic functions; indeed, we can translate and run them without a problem. However, verification of monadic functions is generally eased by providing a library of theorems and notational support, and building this up is a time-consuming undertaking (and even getting the full suite of notation we wanted would have been challenging in the Coq version we were using at the time). Because no other code depended on these monadic `Bag` functions, there were, in our estimation, other, more pressing ways to advance the usage of `hs-to-coq` than building such a library. Thus, we decided that verifying monadic functions will remain out of scope for `hs-to-coq` until such time as better support arrives or we write it ourselves.

Finally, there are three functions on `Bag`s that we did not verify because they referred to code that we did not translate.

`isSingletonBag`: This function tests if a `Bag` contains exactly one element.

`partitionBagWith`: This function splits a `Bag` in two pieces, like `partition`; instead of a predicate, however, it uses a function `p :: a -> Either b c`, and produces one bag with all the `Left` results and one with all the `Right`s.

`mapAccumBagL`: This function corresponds to the function `mapAccumL` for lists. It has the type signature

```
mapAccumBagL :: (acc -> x -> (acc, y))
             -> acc
             -> Bag x
             -> (acc, Bag y)
```

One way to understand it is as `mapM` (or `traverse`) for the state monad using `acc` as the type of the state. It applies the first parameter to each element of the bag, which determines the structure of the output `Bag y`; additionally, for the first element, the initial `acc` value is passed, and for every successive element, the `acc` value generated by the previous one is passed.

5.3.4.1. *New `Bag` operations.* There are also three `Bag` operations that were added in GHC 8.2, which we have not verified yet.

`allBag`: As discussed above, this function checks that every value in the `Bag` satisfies the given predicate.

`concatMapBag`: A combination of `mapBag` and `concatBags`; applies a function that returns a `Bag` to every element of a `Bag`, and concatenates the results.

`mapMaybeBag`: A combination of `mapBag` and `catBagMaybes`; applies a function that returns a `Maybe b` to every element of a `Bag`, and produces a `Bag` consisting of the contents of all the `Just` results.

5.3.5. What did we learn? Verifying the `Bag` module was an excellent early project to help us understand how to use and further develop `hs-to-coq`; we can see that here, as this is the same reason it functioned well as an introductory example in Chapter 2. In particular, in addition to learning about the correctness of this specific Haskell code, we learned some more general things.

First, working with the `Bag` module was an early foray into verifying abstract data types; we did much more work on this with `containers`, which we are about to see in the next chapter (Chapter 6). However, some themes carry through; for example, we implemented our specifications of correctness by providing a semantics for the `Bag` type, associating it with a type that was easier to work with.

We also saw the challenges of specification design: we had to differentiate between well-formedness and correctness, we had to trust certain comments (the invariants) but found that others were wrong (the claim that `Bag`s were unordered, or the specifics of when the two implementation of `foldBag` were the same). This is of course a perennial challenge for all verification; the challenges have a slightly different nature when verifying existing code, but this is still not unique to `hs-to-coq`.

Lastly, this work was the first time we verified a piece of GHC, and the first time we had to slice a portion of GHC out of the rest so that we could work on it. Because our eventual goal was to verify the more compiler-specific aspects of GHC, this was valuable practice for the work we did on that later, which we will see two chapters

from now (Chapter 7). Because the `Bag` module was largely self-contained, this was an easier project, but it helped situate us in that environment when we started on that much more complex task.

CHAPTER 6

"Ready, `Set`, Verify!"

In "Ready, `Set`, verify! Applying `hs-to-coq` to real-world Haskell code (experience report)" (Breitner et al., 2018), we verified the `containers` library, and in particular sets and maps. We extended `hs-to-coq`'s reach to large libraries, and relaxed our adherence to only interacting with total code. While we did not verify "truly" partial functions, we extended `hs-to-coq` to work with "internal partiality" in the implementation of what the API consumer sees as total functions. We also had to seek out specifications from wide-ranging sources to ensure that when we verified `containers`, the result was meaningful. As part of this effort, I used `hs-to-coq` in a particularly novel way: to translate a QuickCheck test suite into a Coq "proof suite", allowing us to formally verify the QuickCheck properties in Coq (Section 6.2). Happily, we found no bugs in `containers`, attesting to the correctness of well-maintained Haskell code.

6.1. Data structures

We verified four data structures from the `containers` library:

(1) `Set`, for unordered finite sets of `Ord`erable elements;
(2) `Map`, for unordered finite key-value maps with `Ord`erable keys;
(3) `IntSet`, for unordered finite sets of `Int`s; and
(4) `IntMap`, for unordered finite key-value maps with `Int` keys.

We did not verify `Graph` (finite graphs), `Tree` (rose trees), or `Seq` (finite sequences).

The `Set` and `Map` types are both based on weight-balanced binary search trees; the `IntSet` and `IntMap` types are both based on big-endian Patricia trees (Morrison, 1968; Okasaki and Gill, 1998). Consequently, the details of verifying `Set`s and `Map`s are very similar, as are the details of verifying `IntSet`s and `IntMap`s; symmetrically, since they represent the same data structure, the specifications for `Set` and `IntSet` are more similar, as are the specifications for `Map` and `IntMap`.

In the initial paper (Breitner et al., 2018), we focused on just `Set` and `IntSet`; however, we have since extended our verification efforts to cover all four data structures to varying degrees. My focuses on this project were (1) developing and extending `hs-to-coq` to support our increasing verification-related needs; and (2) developing the QuickCheck "proof suite" technology.

6.1.1. Weight-balanced binary trees. The definition of `Set` is as a binary tree with empty tips whose nodes contain both a value and the size of the tree rooted there:[17]

[17]Extra dashes were removed from the horizontal rules.

```
{--------------------------------------------------------------------
  Sets are size balanced trees
--------------------------------------------------------------------}
-- | A set of values @a@.

-- See Note: Order of constructors
data Set a    = Bin {-# UNPACK #-} !Size !a !(Set a) !(Set a)
              | Tip

type Size     = Int
```

A well-formed `Set` is a binary search tree, so the membership function need only recurse at most once:

```
member :: Ord a => a -> Set a -> Bool
member = go
  where
    go !_ Tip = False
    go x (Bin _ y l r) = case compare x y of
      LT -> go x l
      GT -> go x r
      EQ -> True
```

Looking at the definition of `Set`, we see an `{-# UNPACK #-}` annotation – the `{-# ... #-}` comment form is interpreted as a pragma by the compiler – as well as multiple `!`s. The `{-# UNPACK #-}` annotation is about the internal representation of the `Bin` constructor: it directs the compiler to take the `Size` value and store it directly in the constructor, removing one layer of indirection. To understand the effect of the `UNPACK` pragma, see Figure 6.1. It ends up saving one word of memory for the constructor header, plus it removes one indirection every time the size is accessed. For small values like machine words which can be stored in registers, this is a powerful optimization. The `!` annotations are strictness annotations; when a constructor is evaluated, so are all of its strict fields.

Since `hs-to-coq` does not handle laziness, and does not support infinite (corecursive) data structures without an explicit request via an edit, we can ignore the strictness annotations; since `{-# UNPACK #-}` is explicitly a semantics-preserving optimization that cannot be represented in Coq, we can ignore it.

This sort of tree representation is only efficient to work with if it stays balanced. The `containers` library uses a *size-balanced* representation:

$$s_l + s_r \leq 1 \quad \vee \quad (s_l \leq 3s_r \wedge s_r \leq 3s_l),$$

where s_l and s_r are the sizes of the left and right subtrees, respectively, of a `Bin` node. This ensures that the two subtrees have sizes that stay relatively close, which provides the requisite efficiency guarantees.

As alluded to before, the definition of `Map` will look familiar, now that we've seen `Set`:

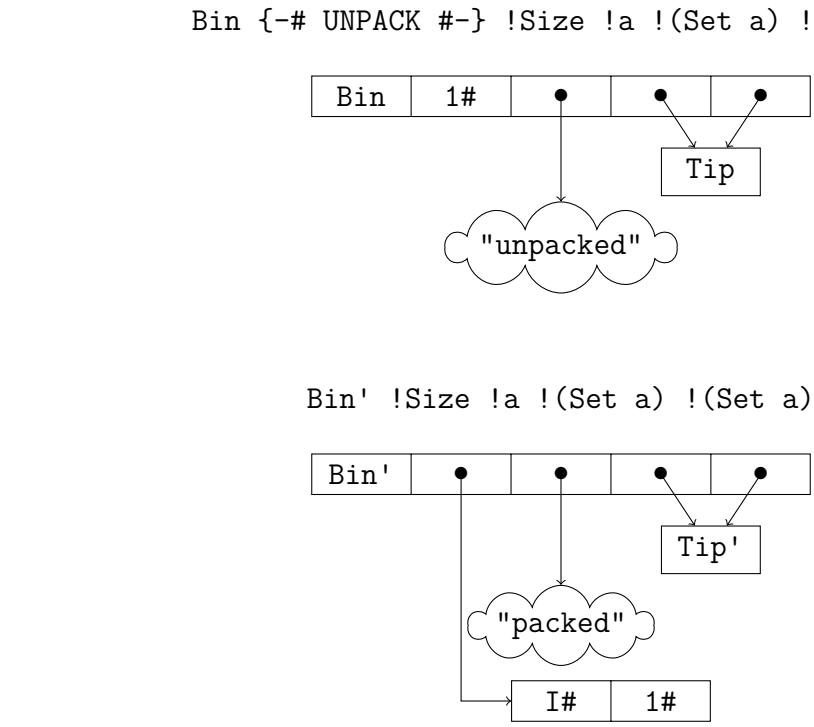

FIGURE 6.1. The effect of an `{-# UNPACK #-}` annotation. The first tree represents the singleton set containing the string `"unpacked"`; the second tree represents the singleton set containing the string `"packed"` if we did not have the annotation. Each rectangle is one word of memory. The • symbol denotes a pointer to another value. A hash-suffixed number represents a machine word; Haskell `Int`s are stored as a constructor (`I#`) wrapping a single machine word. Clouds represent heap values that we do not inspect further. Each constructor (here represented by its name) is stored as a unique one-word tag.

```
data Map k a  = Bin {-# UNPACK #-} !Size !k a !(Map k a) !(Map k a)
              | Tip
```

It is identical to the definition of `Set k`, except for the extra lazy `a` field. This affects the specifications, but means that we were able to focus on just `Set`, instead of conceptualizing `Set` and `Map` separately.

6.1.2. Big-endian Patricia trees. The definition of `IntSet` is rather more involved than that of `Set`. Though also made mostly of `Bin`s and `Tip`s, plus an extra `Nil` constructor for the empty set, the structure of the two is rather different, with many more invariants.

```
-- | A set of integers.

-- See Note: Order of constructors
data IntSet = Bin {-# UNPACK #-} !Prefix {-# UNPACK #-} !Mask
```

```
                  !IntSet !IntSet
-- Invariant: Nil is never found as a child of Bin.
-- Invariant: The Mask is a power of 2.  It is the largest bit
--            position at which two elements of the set differ.
-- Invariant: Prefix is the common high-order bits that all
--            elements share to the left of the Mask bit.
-- Invariant: In Bin prefix mask left right, left consists of the
--            elements that don't have the mask bit set; right is
--            all the elements that do.
            | Tip {-# UNPACK #-} !Prefix {-# UNPACK #-} !BitMap
-- Invariant: The Prefix is zero for the last 5 (on 32 bit arches)
--            or 6 bits (on 64 bit arches). The values of the set
--            represented by a tip are the prefix plus the indices
--            of the set bits in the bit map.
            | Nil

-- A number stored in a set is stored as
-- * Prefix (all but last 5-6 bits) and
-- * BitMap (last 5-6 bits stored as a bitmask)
--   Last 5-6 bits are called a Suffix.

type Prefix = Int
type Mask   = Int
type BitMap = Word
type Key    = Int
```

Testing for membership in an `IntSet` is thus defined in terms of bit manipulation rather than the ordering used in `Set`; this can both pick whether to go down the left or right branch, as well as return `False` early. Each `Tip` also stores multiple values, rather than none at all.

```
-- | /O(min(n,W))/. Is the value a member of the set?

-- See Note: Local 'go' functions and capturing.
member :: Key -> IntSet -> Bool
member !x = go
  where
    go (Bin p m l r)
      | nomatch x p m = False
      | zero x m      = go l
      | otherwise     = go r
    go (Tip y bm) = prefixOf x == y && bitmapOf x .&. bm /= 0
    go Nil = False
```

Once again, the definition of `IntMap` will be familiar:

```
data IntMap a = Bin {-# UNPACK #-} !Prefix
                    {-# UNPACK #-} !Mask
                    !(IntMap a)
                    !(IntMap a)
-- Fields:
--   prefix: The most significant bits shared by all keys in this
--           Bin.
--   mask: The switching bit to determine if a key should follow
--         the left or right subtree of a 'Bin'.
-- Invariant: Nil is never found as a child of Bin.
-- Invariant: The Mask is a power of 2. It is the largest bit
--            position at which two keys of the map differ.
-- Invariant: Prefix is the common high-order bits that all
--            elements share to the left of the Mask bit.
-- Invariant: In Bin prefix mask left right, left consists of the
--            elements that don't have the mask bit set; right is
--            all the elements that do.
              | Tip {-# UNPACK #-} !Key a
              | Nil
```

Down to the comments , this is nearly identical to `IntSet` (except that `Bin` has its fields broken up onto multiple lines), although, because the `Tip` has to map an integer `Key` (a type synonym for `Int`) to a value, it does not have the same multi-element packing optimization that `IntSet` does. Nevertheless, the two types have almost identical structure, with `IntMap` being simpler, and so we could again focus on just `IntSet`.

6.2. From a test suite to a proof suite

In our ravenous search for specifications of `containers`, one source we found useful was its test suite of QuickCheck (Claessen and Hughes, 2000) properties.

6.2.1. What is QuickCheck? QuickCheck is a Haskell library, originally developed by Claessen and Hughes (2000), for *property-based random testing*. To use it, the user defines three different kinds of things:

(1) *Generators*, which generate random values of a specific type;
(2) *Shrinkers*, which take values of a specific type and make them "simpler"; and
(3) *Properties*, which are predicates that are expected to hold for all of their inputs.

The type class `Arbitrary` encapsulates the first two, although it is possible to define generators and shrinkers separately as well.

A QuickCheck property is, typically, a function from some collection of inputs to a `Bool`, that the programmer believes holds on all of their inputs. Supposing that all the input types are instances of `Arbitrary` – that is, they have associated generators

and shrinkers – then the truth of this property can be tested by randomly generating many different inputs and testing that the function always returns `True`. If it ever returns `False`, then a counterexample was found, and the property is false; each input is repeatedly shrunk, using the associated shrinker, and retested in order to find and present a relatively small counterexample.

QuickCheck offers more power than this: the most general type of something testable is `Property`, which captures both generation and returning a boolean. For example, there's a function `forAll` which takes a generator and returns a `Property`, so you can work with specific generators.

6.2.2. How we use QuickCheck. The existence of QuickCheck test suites is incredibly helpful for us, as it is another source of specifications: the library authors believe that each of the properties in the test suite must be true. This gives us a collection of propositions we'd like to transfer to Coq. But, crucially, instead of being in plain text, or thought up anew by us, they are *already in Haskell*. And we have a tool for converting Haskell code into Coq!

We could try just running `hs-to-coq` on these propositions to get Coq functions to `Bool`. But this would fail when dealing with randomness, and with QuickCheck's richer `Property` type. Instead, we take advantage of `hs-to-coq`'s configurability through edits, and change our usual tune: we intentionally try to make the Coq output *different* than the input. In particular, we translate QuickCheck's `Property` to Coq's `Prop`, and translate all the associated combinators required to ensure that these functions become proof-amenable properties. As such, we've gone from a "test suite" to a "proof suite" – or just a "theorem suite" before we've completed the proofs.

6.2.3. Our translation of QuickCheck. To translate QuickCheck, we use four different components:

(1) Hand-written replacement Coq modules for the necessary portions of the Haskell `Test.QuickCheck.*` modules.
(2) Compiler flags for GHC to enable testing mode.
(3) A preamble to use definitions that link to our hand-written replacement modules.
(4) Edits to control the linkage and skip properties and modules we don't want.

The latter three must be redone for every test suite we wish to translate, but the first could in principle be reused (although we have not done so yet).

Item (2) is uninteresting – it merely requires passing the `-DTESTING` flag to GHC, and is only necessary because of the way the `containers` library was defined. This exposes various definitions (for testing purposes) that would otherwise be hidden. Items (1), (3), and (4) are more interesting.

6.2.3.1. *Hand-written QuickCheck replacements.* QuickCheck defines a top-level function

```
quickCheck :: Testable prop => prop -> IO ()
```

This function (or its more-configurable variants) takes any "testable property" and queries it by generating a lot of random inputs. We don't need to use this function – in fact, we can't, as we don't translate `IO`, Coq doesn't support randomness natively, and

we don't provide a Coq PRNG. Instead, we want to make sure that every `Testable` property can be converted to a Coq `Prop`. We can then attempt to prove that any `Testable` property holds.

The `Testable` type class is defined as

```
class Testable prop where
  property :: prop -> Property
  propertyForAllShrinkShow ::
    Gen a -> (a -> [a]) -> (a -> [String]) -> (a -> prop) ->
    Property
```

with the latter function being optional and inessential. This means that the core of being testable is being able to be converted into a `Property`. The `Property` type is abstract, which is perfect for our purposes: we simply want to replace it with Coq's `Prop`. Thus, our Coq definition is instead

```
Class Testable (a : Type) := { toProp : a -> Prop }.
```

That way, just as we would run `quickCheck prop_isOK` in a test suite, we can prove the theorem `toProp prop_isOK` in the corresponding proof suite.

The `Testable` class has several instances, but some of the most important are:

- `Testable Property`, which is just the identity.
- `Testable Bool`, where `True` is a true property/a successful test and `False` is a falsifiable property/a failed test.
- `Testable ()`, which is an always-true property/an always-successful test.
- `(Arbitrary a, Show a, Testable prop) => Testable (a -> prop)`, which lets us test functions by generating random inputs and looking at the result. Applied recursively, this lets us test functions of arbitrary arity.

The function instance is the most interesting, because it controls how quantification works. A function `f :: a -> prop` effectively says that $\forall(\texttt{x} :: \texttt{a}), \texttt{f x}$. It works by requiring that `a` be an instance of `Arbitrary`, which is the type class that permits random generation of values.[18] For our purposes, it might seem that we could get away without this type class – `forall (x : a), toProp (f x)` is guaranteed to itself be a `Prop` without constraints on `a`. But that turns out not to be quite what we want.

The definition of the `Arbitrary` type class is:

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a]
```

The type constructor `Gen` is a monad supporting (sized) random generation, so `arbitrary` generates a random value of type `a`. The `shrink` function is used to generate simpler counterexamples, and so is not relevant for our purposes (where we write proofs and don't need counterexample generation).

The important thing about `arbitrary`, however, is that there is no requirement that it be able to generate *all* inhabitants of `a`. While for many types it does, sometimes it may be constrained. For example, the QuickCheck library contains many newtypes,

[18] It also requires `Show`, which is just for printing the output and so not relevant to our purposes.

such as `newtype Positive = Positive { getPositive :: a }`, whose `Arbitrary` instances are specifically intended to be constrained. Or, a type – such as, say, `IntSet` – may only want to generate well-formed values.

In any event, we want to allow `Arbitrary` to continue to manage these sorts of constraints. In fact, anywhere we could generate a random value, that's not what we want to do – we will instead want to *validate* that any arbitrary input value *could have been* generated by that generator. We thus redefine the `Gen` type to be

```
Record Gen a := MkGen { unGen : a -> Prop }.
Arguments MkGen {_} _.
Arguments unGen {_} _ _.
```

For example, take the generator `choose :: Random a => (a, a) -> Gen a`, which generates a random value in the given range (inclusive). Our handwritten Coq definition of `choose` is instead

```
Definition choose {a} `{GHC.Base.Ord a} (range : a * a) : Gen a :=
  MkGen (fun x => (fst range GHC.Base.<= x) &&
                  (x GHC.Base.<= snd range)).
```

This means that we can then simply leave `Arbitrary` mostly alone:

```
Class Arbitrary (a : Type) := { arbitrary : Gen a }.
```

Nevertheless, thanks to our redefinition of `Gen`, its behavior has completely changed. And we can use this changed behavior to give the `Testable` instance for functions:

```
Definition forAll {a prop} `{Testable prop}
                  (g : Gen a) (p : a -> prop) : Prop :=
  forall (x : a), unGen g x -> toProp (p x).
Arguments forAll {_ _ _} / _ _.

Instance Testable_fn
  {a prop} `{Arbitrary a} `{Testable prop} : Testable (a -> prop) :=
  { toProp := forAll arbitrary }.
```

QuickCheck also features various combinators for operating on `Testable` properties, such as the conjunction-of-properties operator `(.&&.)`, which has the type

```
(Testable prop1, Testable prop2) => prop1 -> prop2 -> Property
```

and requires both its arguments to hold. We can define these combinators simply as their Coq analogues in `Prop`: Haskell `(.&&.)` becomes Coq `/\`, Haskell `(.||.)` becomes Coq `\/`, and so on.

We did not define many instances of `Arbitrary`, just those that were needed to put the proof suite for `IntSet` together. The instances we provided were for `bool`, `Int` (which is `Z` in our Coq translation), `Word` (which is `N`), `list`, and `IntSet`. The first three instances are all essentially the same:

```
Instance Arbitrary_bool : Arbitrary bool :=
  { arbitrary := MkGen xpredT }.
```

(The predicate `xpredT` is from SSReflect, and is the always-true predicate).

The other two instances are somewhat more interesting. Lists need to propagate the `Arbitrary` instance for their contents:

```
Instance Arbitrary_list {a} `{Arbitrary a} : Arbitrary (list a) :=
  { arbitrary := MkGen (Coq.Lists.List.Forall (unGen arbitrary)) }.
```

And `IntSet`s need to be well-formed:

```
Instance Arbitrary_IntSet
    : Test.QuickCheck.Property.Arbitrary Data.IntSet.Internal.IntSet
    :=
  { arbitrary := Test.QuickCheck.Property.MkGen IntSetProofs.WF }.
```

6.2.3.2. *A custom preamble.* Our custom preamble is simple: a few module `Require` statements, one notational definition, the aforementioned `Arbitrary IntSet` instance, and

```
Coercion is_true : bool >-> Sortclass.
```

as can be found in SSReflect.

One of the imported modules is `IntSetProofs`, solely because we need to be able to refer to `WF` in the `Arbitrary IntSet` instance. Interestingly, this posed a problem: the required dependency graph was very different from usual. Dependency-wise, all the other files in `containers` are translated "first", then we write specifications that depend on those definitions, and then we write proofs that depend on those specifications. Put another way, the generated Coq is upstream from the hand-written Coq. But here, we were translating the specification, and so needed to put the translated Coq downstream from the hand-written code! Thankfully, since `hs-to-coq` doesn't check the Coq for correctness, this posed no problems during the translation, but it did require reworking how compilation worked for the Coq code.

6.2.3.3. *Edits for working with QuickCheck.* Unusually for working with `hs-to-coq`, most of the work lay in hand-written modules (Section 6.2.3.1), and not in the edits. Most of the edits were simple:

- redefine number types to be `N`, and switch up operations on them, as per usual;
- skip modules we weren't using, including QuickCheck modules that we were replacing;
- set the name of the module to something other than `Main`;
- skip the `Arbitrary` class, since its instances would be all wrong and need to be hand-written;
- skip the `main` action that runs the tests;
- skip the unit tests; and
- skip the properties that we didn't want to prove.

For the most part, the properties we skipped were those that tested functions we didn't have, although one involved a local fixpoint we did not want to handle.

6.2.4. Our translated proof suite. Once we have this translation, we take each property we wish to work with and prove it as a theorem. The properties all have names like `prop_Member`, `prop_InsertDelete`, `prop_UnionComm`, and so on (as is generally the custom when working with QuickCheck), so for each of these we prove the corresponding theorem: `Theorem thm_Member : toProp prop_Member`, etc. For example,

```
prop_UnionComm :: IntSet -> IntSet -> Bool
prop_UnionComm t1 t2
  = (union t1 t2 == union t2 t1)
```

from `containers` becomes (reformatted and with module names elided)

```
Definition prop_UnionComm : IntSet -> IntSet -> bool :=
  fun t1 t2 => (union t1 t2 == union t2 t1).
```

When we wish to prove it, we prove the theorem

```
Theorem thm_UnionComm : toProp prop_UnionComm.
```

which is equivalent to

```
Theorem thm_UnionComm :
  forall s1 : IntSet, WF s1 -> forall s2 : IntSet, WF s2 ->
  prop_UnionComm s1 s2.
```

(as can be seen, up to α-equivalence, by applying the `simpl` tactic).

The proofs themselves leverage the other results that were proved about `IntSet`, which made them simpler to reason about, although there is nothing intrinsically necessary about this – it is simply that we started with the other proofs, and came to the QuickCheck properties second. One could use this same technique first, and then figure out what general theories (such as `Sem`) were necessary for these proofs. Indeed, unsurprisingly, I had to prove additional lemmas about the behavior of `IntSet`s to complete all the proofs, and in particular to work with SSReflect.

Verified properties. All in all, we proved 34 QuickCheck properties correct.

`thm_Valid`: Asserts that all `arbitrary IntSet`s are also `valid`. The `valid` function is a predicate, defined only for testing purposes, that captures the invariants that should hold of a tree. We already know that `validity` implies `WF`; our translation of this property restates that, so its proof is not complicated given that result.

`thm_EmptyValid`: Asserts that the empty `IntSet` is `valid`.

`thm_SingletonValid`: Asserts that `singleton x` is always `valid`.

`thm_InsertIntoEmptyValid`: Asserts that `insert x empty` is always `valid`.

`thm_Single`: Asserts that `insert x empty` is the same as `singleton`.

`thm_Member`: Asserts that `fromList` preserves membership, turning `elem` on lists into `member` on `IntSet`s.

`thm_NotMember`: Asserts that `fromList` preserves nonmembership, turning `notElem` on lists into `notMember` on `IntSet`s.

`thm_InsertDelete`: Asserts that `delete k . insert k` is the identity, as long as `k` was not already present. Also asserts that the result is `valid`.

thm_MemberFromList: Similar to `thm_Member` and `thm_NotMember`, asserts that if a list is turned into an `IntSet`, then all its elements are members of the result, and nonelements are not (here, the nonelements are the negations of a list of positive numbers).

thm_UnionInsert: Asserts that `union`ing a set with `singleton x` is the same as simply using `insert x`, and that the result is `valid`.

thm_UnionAssoc: Asserts that `union` is associative.

thm_UnionComm: Asserts that `union` is commutative.

thm_Diff: Asserts that the `difference` of two `IntSet`s made with `fromList` is, when converted back to a list, the same as taking their list-difference, up to order and repetition; also asserts that the result is `valid`.

thm_Int: As `thm_Diff`, but with respect to `intersection`.

thm_disjoint: Asserts that two `IntSet`s being `disjoint` is equivalent to their `intersection` being empty (i.e., being `null`).

thm_List: Asserts that converting from and back to a list is the same as sorting that list and dropping repetitions (i.e., that `toAscList . fromList` is the same as `sort . nub`).

thm_DescList: As `thm_List`, but uses `toDescList` and `reverse`s the sorted list.

thm_AscDescList: Asserts that `toAscList` is just `reverse . toDescList` for any `IntSet`s constructed with `fromList`.

thm_Prefix: An invariant that holds on the guts of an `IntSet`: that the prefix bits are correct for all the members of an `IntSet`, and so recursively on its tree structure.

thm_LeftRight: An invariant that holds on the guts of an `IntSet`: that the bitmask is correctly unset for all the elements of the left-hand side of the `IntSet`'s tree structure, and is correctly set for the right-hand ones.

thm_isProperSubsetOf2: Asserts that $A \subsetneq A \cup B \iff A \neq A \cup B$, in terms of `isProperSubsetOf` and `union`.

thm_isSubsetOf2: Asserts that $A \subseteq A \cup B$, in terms of `isSubsetOf` and `union`.

thm_size: Asserts that the `size` on an `IntSet` is the same as both (a) the result of using `foldl'` to compute the size directly, and (b) the result of taking the list length of `toList`.

thm_ord: Asserts that the ordering on `IntSet`s is the same as the ordering on their `toList`s.

thm_foldR: Asserts that `foldr` can be used to reconstruct a list; specifically, asserts that `foldr (:) []` is the same as `toList`.

thm_foldR': As `thm_foldR`, but about `foldr'`.

thm_foldL: Asserts that `foldl` can be used to reconstruct a list, but backwards; specifically, that `foldl (flip (:)) []` is the same as reversing `toList`.

thm_foldL': As `thm_foldL`, but about `foldl'`.

thm_map: Asserts that `map id` is `id`.

thm_split: Asserts that `split` is correct. The result of `split i s` is supposed to be a pair of sets, the first containing all the elements of `s` less than `i` and the second containing all the elements of `s` greater than `i` (with `i` itself gone). This theorem proves that, and also that the resulting sets are `valid`.

thm_splitMember: As `thm_split`, but about `splitMember`; the difference is that in addition to two sets, `splitMember` also returns a boolean saying whether or not `i` had originally been a member of `s`.

thm_splitRoot: Asserts that `splitRoot`, a function exposed solely for testing and debugging purposes, is correct. The only guarantees made (and thus proved) for `splitRoot : IntSet -> IntSet` are that the resulting sets are in ascending order and that their union is the original `IntSet`.

thm_partition: Asserts that `partition odd` correctly returns two `valid IntSet`s that union to the original input, the first containing only odd `Int`s and the second containing only even `Int`s.

thm_filter: Asserts that `filter odd` and `filter even` return `valid` results that are equal to the results of `partition odd`.

Missing properties. There were also 15 properties I did not turn into theorems. I skipped 12 of these, and translated but did not prove 3 of them.

The twelve properties I skipped were:

prop_LookupLT: This property, which tests `lookupLT`, was implemented in terms of the local function `test_LookupSomething`, which was implemented in terms of the partial functions `last` and `head`. We did not translate these partial functions.

prop_LookupGT: As `prop_LookupLT`, but tests `lookupGT`.

prop_LookupLE: As `prop_LookupLT`, but tests `lookupLE`.

prop_LookupGE: As `prop_LookupLT`, but tests `lookupGE`.

prop_Ordered: This property tests the behavior of `fromAscList` against `fromList`; the two are supposed to be the same, except that the former only works if the list is in monotonically increasing order. We could not translate `fromAscList`, as it was defined in terms of nonstructural mutual recursion (in `fromDistinctAscList`, which is the same but assumes unique elements), so we could not translate this property.

prop_fromList: This property tests that `fromList` produces the same (`valid`) result as `fromAscList`, `fromDistinctAscList`, and `foldr insert empty`; just as with `prop_Ordered`, this means it refers to two functions we could not translate.

prop_findMax: This property tests the partial `findMax` function which, being partial – it returns the largest element of the set, and crashes if the set is empty – we could not translate.

prop_findMin: As `prop_findMax`, but about `findMin`.

prop_readShow: This property tests that `read . show` is `id`; we do not translate the `Read` and `Show` type classes.

prop_maxView: This property tests the `maxView` function, which is analogous to `uncons`, returning `Just` the maximal element and the rest of the set, or returning `Nothing` if the set was empty. We could not translate this because `maxView` is defined in terms of a local partial function, or because `prop_maxView` is defined in terms of the partial list `maximum` function.

prop_minView: As `prop_maxView`, but about `minView` (and `minimum`).

prop_bitcount: We did not translate this function, which tests an operation that counts the bits set in a `Word`, because it was defined in terms of a local, nonstructural fixpoint.

The three properties I translated but did not prove were:

prop_MaskPow2: This property is actually *false* as a theorem. It tests that the bitmasks in a well-formed `IntSet` are all powers of two, but it does so by checking that they come from a set of all powers of two from 2^0 through 2^{63}. Since our translation of `IntSet` is in terms of unbounded natural numbers, this fails for sets containing larger values; in particular, `fromList [:: 2^64-1; 2^64]%N` is a concrete counterexample. I was able to prove this property up until I needed to prove membership in that set.

While validating this counterexample, I discovered a bug in our implementation of `base`, having to do with some instances of the `Enum` type class. The Haskell list `[a..b]` is the same as `enumFromTo a b`, and is a list from `a` through `b`, inclusive; for example, `[2..4] == [2,3,4]`. This is a method from the `Enum` type class, and so is available for a variety of types. Our Coq implementation of `enumFromTo` for `N`, however, is implemented in terms of `seq`, which has different behavior: `seq a b` is a length-`b` sequence starting at `a`, so `enumFromTo 2%N 4%N = [:: 2;3;4;5]`! We also implemented the instance for `nat` incorrectly, although that did not manifest here; that instance simply left off the last value, so that `enumFromTo 2%nat 4%nat == [:: 2;3]`. However, the instance for `N` was used to create the set of powers of two mentioned above, and so meant that the set of powers of two only included the ones up through 2^{62}, and did not include 2^{63}.

prop_isProperSubsetOf: This property tests that `Data.IntSet.isProperSubsetOf` is correct by comparing it to `Data.Set.isProperSubsetOf`. We did not prove any theorems about `Data.Set.isProperSubsetOf`, so we did not try to prove that this property was a theorem.

prop_isSubsetOf: As `prop_isProperSubsetOf`, but uses `Data.IntSet.isSubsetOf` and `Data.Set.isSubsetOf`.

6.2.5. What did we learn? Translating the QuickCheck properties was a novel use of `hs-to-coq`; to date, translating QuickCheck properties is the only time we have used `hs-to-coq` to *change* the meaning of the translated code. One lesson that we took home from this experience is that `hs-to-coq` is well suited for that; the immense configurability of edits made this process smooth sailing. However, it did not integrate into our workflow; we assumed that all translated code was one entity that could be compiled, and then that our proofs were a another library that could be compiled against it. By using the Haskell test suite as a live source of specifications, we blurred this distinction. Already, we mixed translated and hand-written modules for libraries themselves; with this, we needed to mix translated and hand-written specification modules.

The only reason `hs-to-coq` could be useful for generating specifications, though, is that the Haskell test suite was strong enough to cover a lot of `IntSet`'s behavior. It is a testament to both the power of testing – and in particular, though not exclusively,

property-based random testing – that we found no bugs in any part of `containers` that we verified. We were not surprised by this – test suites are known to be strong enough to provide reliable guarantees, since people have been using `containers` for years without problems. In fact, the `containers` test suite is strong enough that the only serious bug found in the library's changelog deals with a function (`Data.IntMap.restrictKeys`) whose tests accidentally weren't run with the rest of the test suite! (Smalley, 2017) Yet this translation was still valuable. Why?

Recall that, as part of the DeepSpec project, we want to produce specifications that are rich, two-sided, formal, and live. As always, formal and live are automatic in our setting, since we are only considering Coq proofs about artifacts generated by `hs-to-coq`. But rich and two-sided are not automatic. A carefully-designed test suite, like the one for `containers`, should be both rich and two-sided: it should test detailed behaviors (rich) that guarantee users will not experience bugs (two-sided). By using a battle-hardened test suite as a source for specifications, we guarantee that our specification includes properties are known to be useful by the implementers – and, since the library is popular, *users* – of the library. Moreover, our specification becomes live in an even more interesting way: not only is it *connected* directly to the implementation, this portion of it *comes* directly and automatically from the implementation.

In addition to being two-sided itself, the "proof suite" helps validate our other specifications. Because we wrote the proofs of the QuickCheck properties in terms of our other theorems about `IntSet`, we help ensure that the custom specification we built is two-sided: it both relates to the implementation and is also strong enough to verify these known-useful, believed-to-be-true properties.

Finally, the proof suite also helps us validate the correctness of `hs-to-coq`. Since we have not verified `hs-to-coq` (nor will we), we are always looking for further ways we can gain confidence in its correctness. Here, we used it to successfully prove that a number of passing tests were in fact theorems. While that does not *guarantee* that `hs-to-coq` is correct, it it certainly true that many bugs it could have would prevent us from carrying out these proofs.

We are certainly not the first to connect property-based random testing to proofs (Bulwahn, 2012; Lampropoulos and Pierce, 2018). Our particular approach connects to the paper that introduced random testing (Claessen and Hughes, 2000), where Claessen and Hughes say that they "have designed a simple domain-specific language of *testable specifications* which the tester uses to define expected properties of the functions under test." We are simply taking these testable specifications and making them verifiable. Other approaches to integrating random testing with proofs include QuickChick (Lampropoulos and Pierce, 2018), which provides an integrated property-based random testing environment in Coq, and Isabelle's approach of using property-based random testing (and some other techniques) on all theorems to make the programmer's life easier (Bulwahn, 2012; Blanchette and Nipkow, 2010). However, these techniques apply QuickCheck in the other direction: they take properties that the person writing the specifications already wrote, and help ensure that those properties are true or provide counterexamples if they are not; they go from properties to testing in advance of proofs. Our technique takes believed-true properties that have already

been tested, and converts them to provable form; it goes from tested properties to provable properties, again in advance of proof.

We agree with Claessen and Hughes when they say that they "are convinced that one of the major advantages of using QuickCheck is that it encourages us to formulate formal specifications". Though Claessen and Hughes (2000) appreciated this because it "improv[ed] [their] understanding of [their] programs", we find that another advantage is that it eases formal specification by taking care of some of the work in advance.

CHAPTER 7

"Embracing a Mechanized Formalization Gap"

My most recent work on `hs-to-coq` has focused on GHC itself (specifically, GHC 8.4.3). Verifying GHC was our initial goal at the outset of this project, and `hs-to-coq` has now advanced to the point where we can do so in ways far more interesting than merely looking at a simple data structure (Chapter 2). We are now able to begin the work of verifying the pieces of GHC that are identifiably pieces of a *compiler*. The particular challenges of verifying any portion of GHC lie in its nature as an application: GHC makes heavy use of partial functions, recursive modules, mutable state, unboxed types, and all sorts of other Haskell features that Coq – and `hs-to-coq` – don't support.

To make this codebase amenable to being translated by `hs-to-coq`, I had to extend `hs-to-coq` to support more edits, many of these with the intent of *altering* the functionality of the code. Previously, we have seen how `hs-to-coq` works when our primary goal is to exactly preserve the semantics of the translated code. Here, we have widened our net, embracing more significant alterations of meaning: ignoring partiality, changing data types, slicing out targeted portions of code, and more. These improvements to the edit language are my chief contributions that I present in this chapter, along with an evaluation of their fitness for service by seeing how they were used to verify parts of GHC. Sections 7.5–7.7 discuss three such applications in this project. In order to further contextualize these changes to the edit language, I also discuss the compiler operations of GHC that we translated and verified as part of this project. These efforts were directly supported by my changes to `hs-to-coq`, and serve both as an end in their own right as well as a means of validating the newly-added edits.

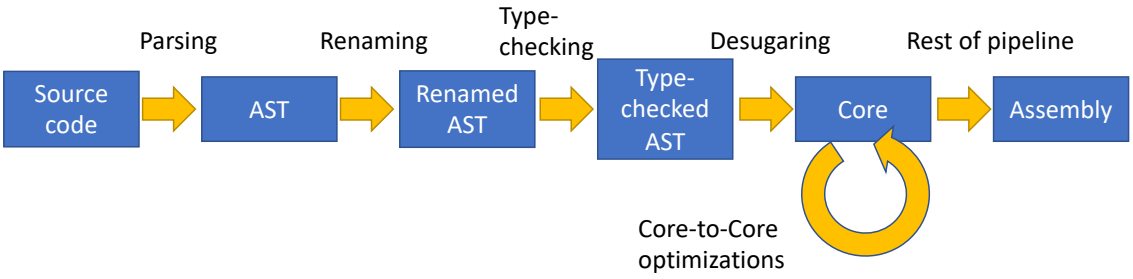

FIGURE 7.1. The internal compilation pipeline of GHC, with details for the front-end.

7.1. The structure of GHC

GHC, like your typical modern compiler, compiles Haskell source code (also referred to as "source Haskell") into assembly/machine code by way of multiple intermediate languages. In particular, in GHC, the bulk of the optimizations are applied to the intermediate language named Core. The full path of intermediate languages is, very roughly (Marlow and Peyton-Jones, 2012):

(1) *Haskell*, the input language.
(2) *Core*, a simpler functional language where optimization happens.
(3) *STG*, a language for the "Spineless Tagless G-Machine" (Peyton Jones, 1992) which is the bridge between the functional and imperative worlds.
(4) *Cmm* (C-minus-minus), a simple low-level imperative language.
(5) *Native code*: either architecture-specific, LLVM, or plain C.

In addition to translating between the different languages, GHC also transforms its internal representation of programs within these different layers; for example, type checking is a transformation that turns a Haskell AST into a different Haskell AST. A diagram of the front-end of the compiler (i.e., the first two languages, where everything is still functional) is presented in Figure 7.1.

Each of these steps has nontrivial correctness constraints, so we needed to focus our attention. We decided to focus on Core; specifically, we focused on optimizations. We wanted to focus on Core for the same reasons the GHC developers want to use it: Core is *simple* and it is *purely functional*. Haskell is purely functional, but very complex – its myriad syntactic cases mean that the data types corresponding to its AST alone have dozens of constructors, and there is no explicit formal semantics of Haskell. Languages such as STG and Cmm, while they may be simple, are not purely functional, so they require different forms of reasoning. Core, then, is just right.

7.2. What is Core?

The Core language is essentially a concrete implementation of System FC, a typed lambda calculus with higher-kinded polymorphism (the System F, or specifically System F_ω, part), algebraic data types, and *coercions* (the C part). Coercions are the feature of System FC that most differentiate it from other lambda calculi, yet we do not need them at all for our work. That said, it still behooves us to introduce them here.

While most of the terms of System FC are standard, it adds a *cast* term: $e \rhd \gamma$ casts the expression e by the *coercion* γ. These coercions are witnesses of equality between two types; if a coercion γ is a witness that the types σ and τ are equivalent, we write this as $\gamma : \sigma \sim \tau$. The typing rule for casts says that if an expression e has type σ, and a coercion γ proves that σ is equal to τ, then $e \rhd \gamma$ has type τ:

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \gamma : \sigma \sim \tau}{\Gamma \vdash e \rhd \sigma : \tau}$$

```
data Expr b
   = Var Id
   | Lit Literal
   | App (Expr b) (Arg b)
   | Lam b (Expr b)
   | Let (Bind b) (Expr b)
   | Case (Expr b) b Type [Alt b]
   | Cast (Expr b) Coercion
   | Tick (Tickish Id) (Expr b)
   | Type Type
   | Coercion Coercion
  deriving Data

data Bind b
   = NonRec b (Expr b)
   | Rec [(b, (Expr b))]
  deriving Data

type Arg b = Expr b
type Alt b = (AltCon, [b], Expr b)

type Id = Var
type CoreBndr = Var
type CoreExpr = Expr CoreBndr
type CoreBind = Bind CoreBndr
type CoreProgram = [CoreBind]
```

```
Inductive Expr b : Type
 := | Mk_Var : Id -> Expr b
    | Lit : Literal -> Expr b
    | App : (Expr b) -> (Expr b) -> Expr b
    | Lam : b -> (Expr b) -> Expr b
    | Let : (Bind b) -> (Expr b) -> Expr b
    | Case : (Expr b) ->  b -> Type_
      -> list ((fun b_ =>
                  (AltCon * list b_ * Expr b_)) b)
      -> Expr b
    | Cast : (Expr b) -> Coercion -> Expr b
    | Mk_Type : Type_ -> Expr b
    | Mk_Coercion : Coercion -> Expr b
with Bind b : Type
 := | NonRec : b -> (Expr b) -> Bind b
    | Rec : list (b * (Expr b)) -> Bind b.

Definition Arg := Expr.
Definition Alt := fun b_ =>
                    (AltCon * list b_ * Expr b_).
Definition Id := Var.
Definition CoreBndr := Var.
Definition CoreExpr := (Expr CoreBndr).
Definition CoreBind := (Bind CoreBndr).
Definition CoreProgram := (list CoreBind).
```

FIGURE 7.2. Haskell (left) and Gallina (right) versions of the Core AST.

This rule is slightly more complex in GHC,[19] but the details do not concern us – we are only concerned with the structure of Core programs.

The last question about coercions is, how do they impact running System FC or Core programs? The answer is simple: they do not! The point of coercions is to be *zero-cost* – in a precise way, the runtime behavior of a System FC program is the same if we erase all the coercions (Sulzmann, Chakravarty, Peyton Jones, and Donnelly, 2007), and this is what GHC does when it compiles Core.

As it is the internal language of a compiler, Core (as opposed to System FC) is defined by the Haskell data types that make it up, which we present on the left-hand side of Figure 7.2. The type `Expr b` is the type of Core terms, and is parameterized by the type `b` of variable bindings. For our purposes, this type will usually be `Var` (also called `Id`). The constructors of this type are:

(1) `Var`, `App`, `Lam`: These three constructors are the basic pieces of any lambda calculus. Lambdas (`Lam`) are defined using an explicitly named representation, which the GHC developers found to be faster despite being more complicated (Peyton Jones and Marlow, 2002). This features heavily in our proofs. Note that the `App` constructor uses the `Arg` type synonym for `Expr`; this is

[19] And even, to a lesser extent, in System FC – the typical presentation uses separate $\vdash$ judgements for typing terms and coercions.

because there are invariants on the data structure, and not every constructor of `Expr` is legal in argument position.

(2) `Case`: This constructor is for pattern-matching, the basic primitive for algebraic data types.[20] The `Case` operation is a variable-binding operation: its first two arguments are the expression to match on (forcing evaluation), and a name to bind the result to. The type the match is being performed on is also recorded, followed by a list of branches of the match (triples of the constructor, the bound variables, and the right-hand side).

(3) `Lit`: This constructor is for literals – numeric literals such as `42`, string literals such as `"Hello, world!"`, and so on.

(4) `Let`: This constructor is a let-expression for binding local variables. The `Bind` type records what variables are bound: either a single variable (`NonRec`) or a recursive group (`Rec`).

(5) `Cast`: This constructor is the aforementioned cast operator which applies coercions.

(6) `Type` and `Coercion`: These constructors are for representing types and coercions, respectively, in the expression language.

(7) `Tick`: This constructor wraps an expression in profiling information, and does not correspond to anything in System FC.

This data type is translated into Coq by `hs-to-coq`, giving the code on the right-hand side of Figure 7.2. Here, we see that the translation is relatively straightforward; the most notable differences are:

(1) We have simply deleted the `Tick` constructor, as we never deal with profiling information.

(2) We have prepended `Mk_` to the `Var`, `Type`, and `Coercion` constructors, since each of these constructors has the same name as a type and Coq only has one namespace.

(3) We have expanded the `Alt` type synonym in the type of the `Case` constructor, since Coq does not allow mutual induction-recursion.

Once we have done this translation, we now have a Coq version of Core. This enables us to proceed with the other significant translation goal of this project: translating the functions from GHC that operate on Core in order to verify them. It is for this that the bulk of our translation edits will be used.

7.3. Disentangling GHC

One broad structural change that we need to apply to GHC in order to perform our verification is to break apart mutual recursion, both in types and, astonishingly, in modules. Even though Core is a relatively simple language, the practical realities of a compiler require a staggering array of supporting types: not simply expressions like `Expr`, and not simply the types and coercions that we do not translate, but also a panoply of forms of metadata, used during program analysis and optimization. These types and their supporting functions are spread across multiple files, but this is a more complex situation than simply having scattered code. These types that define

[20] Unless you prefer to work with eliminator functions, but GHC does not.

Core are *all mutually recursive*, producing a thorny mutually recursive type with *far* more than the two cases of `Expr` and `Bind`. And because the mutually recursive types are spread across multiple files, this gives GHC the dubious distinction of being one of the only Haskell programs that uses mutually recursive *modules*.

The mutual recursion makes the translation of Core difficult not just in size, but also in style. We want to eliminate as much of the mutual recursion as possible in order to ease our proof burden, and we *must* eliminate all uses of mutually recursive modules because Coq does not support them. Our solution to the problem of mutually recursive modules is the most straightforward one: we simply combine them all into a single Coq module, which we call `Core`. We do this by using the **`rename module`** edit (Section 8.6.3), which, when told to give multiple Haskell modules the same Coq name, simply merges them and produces one module as output. This technically accomplishes the goal, but the resulting module is *enormous*, and the types that make up Core are still incredibly mutually recursive. Our solution to that problem involves applying edits to the code in order to simplify things as much as we can.

In general, our foremost goal with the edits is to transform the types to cut out as much of their mutual recursion as possible. This pays dividends twice over: firstly, we get a smaller and simpler type and less code in general. As seen in Figure 7.2, in fact, the eventual mutually-recursive cluster contains only two types, `Expr` and `Bind`. Secondly, by breaking the mutual recursion between the different types, we can break the mutual recursion between the different modules as well: once the types are separated, some of the modules no longer need to be part of the mutually-recursive cycle. This both gives our translated code a more similar structure to the original Haskell code and, again, cuts down on the size of the `Core` module. Once we have done this as much as we can, we also perform further simplification until the situation is manageable, yet retains the essential complexity of the functionality we verify. The result of all this simplification is given in Figure 7.3; we now explain the details.

Our first simplification step is to figure out what code we can omit, with **`skip`** and similar edits (Section 4.1.1). There are large portions of the modules defining Core that we don't care about. As discussed above, we don't need the definitions of types and coercions. We also maintain an invariant that our data structures should contain *no metadata*, as we aren't attempting to verify the parts of any optimizations that interact with the metadata; we are only interested in the expression structure of Core. Lastly, there's a great deal of code, such as pretty-printing, which simply isn't a target of verification. For as much of this as possible, we simply skip it entirely. Skipping is powerful for both cutting the mutual recursion (such as via **`skip constructor`**, Section 8.2.2) and for simply reducing the amount of code; for instance, eliminating metadata helps with the former, and skipping pretty-printing helps with the later.

Sometimes, skipping is too broad of a brush, but we still don't care about the details. In this case, we use axiomatization edits (Section 4.1.2) to simplify the *content* of the types and values, while still leaving them in place. Again, this helps both with splitting out types – axioms neither can nor need to be in a mutually-recursive cycle with anything else – and simplifying the code we are dealing with. We axiomatize whole modules that can become independent of the mutually-recursive cycle using

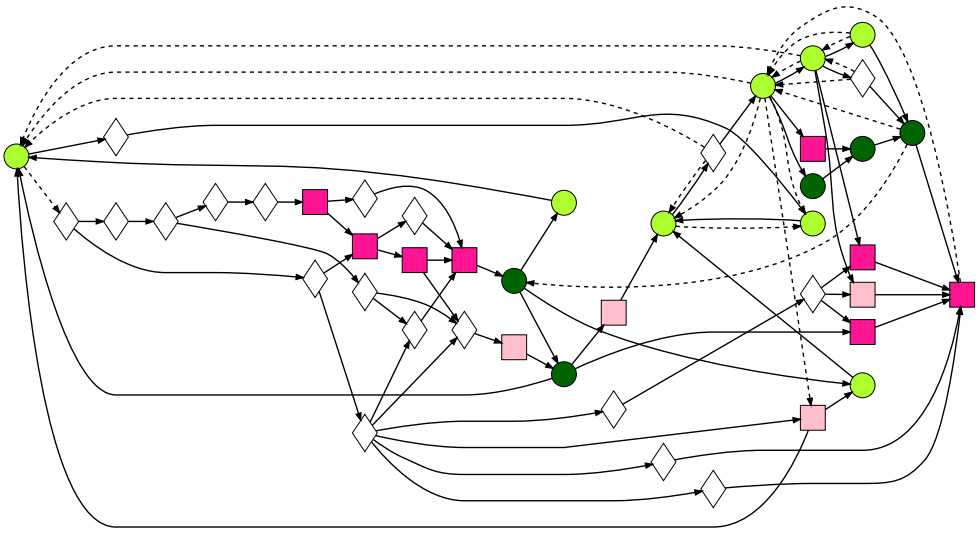

FIGURE 7.3. The dependency graph of the modules that make up Core. Each polygonal node is a Haskell module, and each arrow is a module import. Solid lines are normal imports; dashed lines are mutually-recursive imports (called "source" imports). The shapes and colors of the nodes differentiate the final status of the corresponding module after translation: green circles become part of the translated Coq `Core` module, with darker circles being translated and lighter ones being axiomatized; pink squares become standalone Coq modules that are not part of `Core`, again with darker squares being translated and lighter squares being axiomatized; and white lozenges are skipped and do not appear in our Coq translation. (This figure is due to my collaborator and advisor Stephanie Weirich.)

`axiomatize module` (Section 8.3.1), and axiomatize whole modules that go into `Core` using **`axiomatize original module name`** (Section 8.3.2).

Once we have made these changes, we have accomplished much of the simplification of Core that we wanted, but not all of it. Many more specific edits go into the translation: some in the broader goal of simplification; some in the further service of breaking apart the mutual recursion; and some in the goal of making a workable translation. For example: To further simplify things, we remove the code that computes sets of free type variables, since we don't have types in our translation; we discuss how we do this and the tradeoffs of the approach in Section 7.6. To further help break the mutual recursion, we redefine some types to contain less information; we discuss these simplifications further in Section 7.5. And to produce a workable translation, we have to be very deliberate in how and why we break apart the mutual recursion,

since the unmodified type is not accepted by Coq; we discuss why it isn't and how we fix this in Section 7.5 as well.

7.4. Edits for GHC

As the previous section indicates, the sheer scale of translating GHC, even after restricting ourselves to a fragment thereof, is such that it required adding new functionality to `hs-to-coq`; in particular, it required adding a bevy of new edits. The improvements we made to `hs-to-coq` were inspired both by the unique features of the GHC code base, as well as by our evolving approach to translation. The latter is the more important motivation: as opposed to the work we saw in previous chapters, we are now more willing to modify the semantics, as well as the structure, of the translated code.

In this section, we describe the new features of `hs-to-coq` in more detail. However, to provide context, we first summarize the existing capabilities of `hs-to-coq` edits that we have seen thus far and their typical use in translation.

Building on the existing capabilities of `hs-to-coq` edits. Prior work on translating the `containers` library demonstrated the capabilities of `hs-to-coq` for medium-scale programming (Breitner et al., 2018). At this scale, it was already necessary for users of the tool to define a number of edits in service of mechanical verification. These edits included:

- Removing Haskell features that make no sense in a shallow embedding, including `reallyUnsafePtrEquality#` and `seq`, using **`rewrite`** edits.
- Skipping parts of the code that are irrelevant for verification or are difficult to translate (such as code related to serialization and deserialization), using **`skip`** edits.
- Modifying the representation of integers to avoid reasoning about overflow, using **`rename type`** edits.
- Substituting operations that are difficult to reason about (e.g., bit twiddling functions) with simpler definitions, using **`redefine`** edits.
- Managing recursive functions that are not structurally recursive by either providing hints for the termination proof along with the definition or by deferring those proofs altogether, using **`termination`** edits.

The translation of GHC also requires all of these previously-extant edit forms. However, we found that this functionality was not enough. Therefore, we have added the following new edit forms to `hs-to-coq`. Later in this chapter, we describe in more detail how we make use of these features when simplifying the Core expression data type and its operations.

Constructor skipping. The Core AST, though simpler than source Haskell, carries around a great deal of information that is not germane to our verification goals. Some of this information is in the form of metadata; some of it is in the form of type and coercion variables that we do not analyze. Regardless, since our model of GHC does not concern itself with these properties, we would like to avoid dealing with these concerns. However, the problematic cases are often *sub*cases of other data types – for example, the `Expr` data type for Core contains a case `Tick` strictly for

profiling information, as we see in Figure 7.2. Thus, we added support for a new `skip constructor` edit that eliminates an entire case from data types. For example, the edit `skip constructor Core.Tick` removes the `Tick` constructor from `Expr` and then propagates this information to delete any equation of a function definition or arm of a case statement that matches, directly or indirectly, against this constructor. More dramatically, we use this edit to modify the representation of variables.

In this way, the development of the embedding gives us assurance that our code of interest is independent of particular features of GHC, without doing any proofs. In particular, if the targeted code needed to use the skipped constructor in a fundamental way (e.g., if it were used to construct a value in some operation that could itself be skipped), then the output of `hs-to-coq` would not be accepted by Coq. We can only skip constructors that we can isolate.

We do need to be careful, however – heavy use of `skip constructor` can lead to wildcard cases that no longer match anything, as all the would-be "extra" constructors have been skipped. Because Coq does not permit redundant pattern matches, we also add an edit to manually delete such cases. The edit `skip equation f pat1 pat2 ...` removes the equation matching `pat1 pat2 ...` from the definition of the function `f`.

Axiomatization. The ability to axiomatize definitions and modules was added to `hs-to-coq` directly for this project. There are many definitions in GHC that we don't want to translate for one reason or another. However, sometimes this code is used within the other functions that we want to verify, but not on a code path that is exercised by our proof (for example, in a metadata update). We thus provide an `axiomatize definition` edit, which replaces any value in Haskell with a Coq `Axiom` at the same type. In contrast to a `redefine` edit, axiomatization is more limited – we make fewer assumptions about the behavior of the edited operation. This edit was extremely valuable during the translation of GHC, as it also allowed for incremental verification.

We offer multiple ways to interact with axiomatization. For example, while we are focused on Core, its dependencies transitively reach into huge portions of GHC, and we don't want to deal with all of them. While we can skip some modules (via `skip module`), this isn't always viable. For example, the `FastStringEnv` module declares a type for maps keyed by GHC's `FastString` type. These are used, for example, when manipulating metadata for data type constructors, but this is not an operation we need to concern ourselves with to verify properties of variables. We can thus `axiomatize module FastStringEnv`, which leaves the type definitions intact and automatically axiomatizes every definition in the module as per `axiomatize definition`.

We also want to make use of axioms to replace *type* definitions. As type definitions do not have kind annotations, we cannot automatically generate axioms; we instead use the `redefine` or `rename` edits to replace one definition with another. For example, the edit `redefine Axiom DynFlags.DynFlags : Type` replaces the `DynFlags` type, a record of configuration options, with an opaque axiom.

While being able to axiomatize Haskell definitions is important, it does have the potential to introduce inconsistency – if we axiomatized the Haskell definition `undefined :: a`, we would be able to prove any theorem we wanted. As a result we

need to examine the functions we axiomatize; however, in GHC, most functions are not fully polymorphic and return inhabited types. In particular, this meant that all of the functions we needed to axiomatize returned inhabited types.

The use of axiomatization was specifically important when extracting a slice of GHC. However, even though we automatically axiomatize many definitions in our development, we almost never manually add axioms about their properties. This is because we chiefly use this feature for what is effectively "dead code": code that is alive and used in GHC, but that is used in portions of the translation (for instance, error message generation) that we didn't prove anything about.

Mutually recursive modules. As we discussed in Section 7.3, GHC, nearly uniquely among Haskell programs, makes significant use of *recursive* modules; most of the modules that define Core are part of a single mutually-recursive cycle. This is not a feature supported by Coq, so as part of the translation, we had to introduce edits that combine multiple source modules (both translated and axiomatized) into a single target. We use this facility to create the module `Core`, which contains the definition of the abstract syntax of the Core intermediate language.

Mutual recursion edits. The Core AST that we saw in Figure 7.2 is defined via the mutually inductive types `Expr` and `Bind`. Typically, operations that work with either of these data structures are perforce mutually recursive. For example, consider the `exprSize` operation, shown in the left hand side of Figure 7.4; it is mutually recursive with the analogous function `bindSize`. These functions compute the size of an expression and a binder, respectively.

Coq can natively show the termination of mutually defined fixed points, as long as they are each recursive on one of the mutually recursive types and make recursive calls to each other on strict subterms of their arguments. One important pattern this naive treatment of termination prohibits is "preprocessing" – recursion that goes through an extra function which simply forwards to one of the recursive functions. This is something we see in as simple a definition as that of `exprSize` and `bindSize`. Along the way, the function `altSize` is called, which simply unpacks a tuple and recurses into `exprSize` and `bindSize`. From Coq's perspective, that means *all three* of these functions are mutually recursive, and one of them has as its only argument a tuple. And there's a fourth function, `pairSize`, which has the same problem.

Since a tuple isn't a mutually recursive inductive data type, for Coq to accept the definitions of `exprSize` et al., we must *inline* the definitions of `pairSize` and `altSize` into the mutually recursive definitions that use them. To tell `hs-to-coq` to do this, we use the `inline mutual` edit:

```
inline mutual CoreStats.pairSize
inline mutual CoreStats.altSize
```

This results in the Coq definition on the right in Figure 7.4, as well as new free-standing *non*-recursive definitions of `pairSize` and `altSize` which are the same as their (local) `let`-bound definitions.

Partiality. As we saw earlier, our previous work either avoided partial operations altogether (Chapter 5) or attempted to isolate them behind total interfaces (Chapter 6). That isn't possible with GHC – many more operations may fail, for a number of

```
exprSize :: CoreExpr -> Int
-- ^ A measure of the size of the expressions,
-- strictly greater than 0
exprSize (Var _)         = 1
exprSize (Lit _)         = 1
exprSize (App f a)       = exprSize f + exprSize a
exprSize (Lam b e)       = bndrSize b + exprSize e
exprSize (Let b e)       = bindSize b + exprSize e
exprSize (Case e b _ as) =
  exprSize e + bndrSize b + 1 + sum (map altSize as)
exprSize (Cast e _)      = 1 + exprSize e
exprSize (Tick n e)      = tickSize n + exprSize e
exprSize (Type _)        = 1
exprSize (Coercion _)    = 1

bindSize :: CoreBind -> Int
bindSize (NonRec b e) = bndrSize b + exprSize e
bindSize (Rec prs)    = sum (map pairSize prs)

pairSize :: (Var, CoreExpr) -> Int
pairSize (b,e) = bndrSize b + exprSize e

altSize :: CoreAlt -> Int
altSize (_,bs,e) = bndrsSize bs + exprSize e
```

```
Definition exprSize : CoreExpr -> nat :=
  fix exprSize (arg_0__ : CoreExpr) : nat :=
    let altSize (arg_0__ : CoreAlt) : nat :=
          let 'pair (pair _ bs) e :=
                arg_0__ in
          bndrsSize bs + exprSize e in
    match arg_0__ with
    | Mk_Var _ => 1
    | Lit _ => 1
    | App f a => exprSize f + exprSize a
    | Lam b e => bndrSize b + exprSize e
    | Let b e => bindSize b + exprSize e
    | Case e b _ as_ =>
       ((exprSize e + bndrSize b) + 1) +
       sum (map altSize as_)
    | Cast e _ => 1 + exprSize e
    | Mk_Type _ => 1
    | Mk_Coercion _ => 1
    end
with bindSize (arg_0__ : CoreBind) : nat :=
   let pairSize
           (arg_0__ : (Var * CoreExpr)%type)
             : nat :=
         let 'pair b e := arg_0__ in
         bndrSize b + exprSize e in
   match arg_0__ with
   | NonRec b e => bndrSize b + exprSize e
   | Rec prs => sum (map pairSize prs)
   end
for exprSize.
```

FIGURE 7.4. Mutual recursion in Haskell (left) and Gallina (right).

reasons. At the same time, we only really care about *total* compiler code. If a compiler fails to produce an output for some input, then a compiler correctness proof says nothing.

One source of partiality comes from the use of GHC's operation for signaling a run-time error (i.e., a compiler bug) – the function `Panic.panic`. We cannot translate this function, since it actually throws an exception (using `unsafeDupablePerformIO`, no less). Instead, we axiomatize this operation as follows:

```
Axiom panic : forall {a} `{GHC.Err.Default a}, GHC.Base.String -> a.
```

The `Default` class, which we introduced when verifying `containers` (Breitner et al., 2018) and I previously discussed in Section 3.8, is a class for types with at least one inhabitant, which is used with opacity to avoid any dependence on what the specific inhabitant is. Instances of it are mostly autogenerated by `hs-to-coq`. This constraint therefore enforces that `panic` can only be called with a return type that is known to be inhabited, ensuring that it does not introduce unsoundness. Although `panic` does not terminate the entire Coq program as it does in Haskell, arriving at it in a proof terminates our ability to reason about the code. Therefore, proving properties about code that uses `panic` also increases our confidence that it will not be triggered on that code path.

Partiality turns out to be important for translating GHC, much more so than we realized going in. For example, Haskell can define record selectors for single constructors of data types with multiple branches; these record selectors are thus necessarily partial.

While the `Default` class is not new, we made much more significant use of it than before when translating GHC due in particular to GHC's pervasive use of partial record selectors. As a result, we sometimes need to add `Default` constraints to types where they weren't already present. To guide this translation, we use the new **`set type`** edit, which allows us to change the type of a definition to a new type of our choosing. It is always safe to use this edit, as Coq's typechecker will prevent us from assigning an inconsistent type to a definition.

Type inference. Haskell's ability to perform type inference is significantly stronger than Coq's, particularly for program fragments that remain (as many do) within the bounds of Hindley-Milner type inference. For the most part, Coq's type inferencer is powerful enough, when combined with the presence of type annotations on top-level bindings, to infer all the types we need. There are, however, occasional exceptions. One subtle case is that Coq cannot infer a polymorphic type without explicit binders, as it cannot insert binders for type variables automatically.

In order to work around this, we augmented `hs-to-coq` in two ways. The most direct way is not an edit; instead, we taught `hs-to-coq` to annotate the binders of a fixpoint with their types. In other words, it will replace

```
let f : A -> B := fix f x := ... in ...
```

with

```
let f : A -> B := fix f (x : A) : B := ... in ...
```

Without this transformation, Coq's type inferencer would sometimes fail to infer the type of a fixpoint, as the type information was just too far away.

This change solves most cases of type inference failure, but not quite all of them. To address some corner cases of type inference failure, we also used the above **`set type`** edit. While this edit has a broader purpose, its use in this case is that it allowed us to monomorphize local functions that could have been polymorphic but were only ever used at one type. In particular, this monomorphization was relative to the containing function, as though providing a type signature in Haskell with `{-# LANGUAGE ScopedTypeVariables #-}`. Most of a time, this situation is not an issue, as the context of the function provides enough information. However, in combination with **`inline mutual`**, we can produce dead code that Coq sees as ambiguous.

To see this type inference failure in action, as well as how to use **`set type`** to fix it, I present some simple Haskell code that exhibits the same problem we see in GHC. We define a function, `concatFsts`, that takes a list of pairs whose first component is a string and concatenates those strings:

```
concatFsts :: [(String,a)] -> String
concatFsts = go where
  go []     = []
```

```
  go (t:ts) = goTuple t ts

  goTuple (s,x) ts = s ++ go ts
```

Notice that `goTuple` could instead have the more polymorphic type

```
forall b. (String,b) -> [(String,a)] -> String
```

(where `a` refers to the `a` bound in the top-level type signature of `concatFsts`). This is the crux of the problem.

To translate this code into Coq, we need to use `inline mutual` to ensure that `goTuple` is handled correctly:

```
in Mod.concatFsts inline mutual goTuple
```

This produces the following translated Coq code:

```
Definition concatFsts {a}
    : list (GHC.Base.String * a)%type -> GHC.Base.String :=
  let fix go arg_0__
            := let goTuple arg_3__ arg_4__ :=
                 match arg_3__, arg_4__ with
                 | pair s x, ts => Coq.Init.Datatypes.app s (go ts)
                 end in
               match arg_0__ with
               | nil => nil
               | cons t ts => goTuple t ts
               end in
  let goTuple :=
    fun arg_3__ arg_4__ =>
      match arg_3__, arg_4__ with
      | pair s x, ts => Coq.Init.Datatypes.app s (go ts)
      end in
  go.
```

Thanks to `inline mutual`, we now have two definitions of `goTuple` inside `concatFsts`: one inside `go`, and one free-standing. The former is fine; its type is inferred to be constrained based on the surrounding context. However, the latter is a problem. Thanks to `inline mutual`, it is not linked to the definition inside `go`. And because we never actually use `goTuple` in the definition of `concatFsts` *outside* of `go`, this outside definition *can* be polymorphic, and Coq cannot constrain it: it produces the error "Cannot infer the 2nd argument of the inductive type (`prod`) of this term", referring to the use of `arg_3__` in the `match` inside the free-standing definition of `goTuple`. To fix this, we could just delete that definition of `goTuple`; without an obvious way to do that with edits, however, we settle for monomorphizing the type. If we set the type of `goTuple` to have just the specific type we use it at, all our problems disappear. The full slate of edits is thus

```
in Mod.concatFsts inline mutual goTuple
in Mod.concatFsts set type goTuple : ↩
  (GHC.Base.String * a) -> ↩
  (list (GHC.Base.String * a) -> ↩
    GHC.Base.String)
```

The resulting Coq code differs from the previous code only in the presence of the extra type annotations on (both copies of) `goTuple`:

```
Definition concatFsts {a}
    : list (GHC.Base.String * a)%type -> GHC.Base.String :=
  let fix go arg_0__
            := let goTuple (arg_3__ : GHC.Base.String * a)
                   (arg_4__ : list (GHC.Base.String * a))
                    : GHC.Base.String :=
                     match arg_3__, arg_4__ with
                     | pair s x, ts => Coq.Init.Datatypes.app s (go ts)
                     end in
               match arg_0__ with
               | nil => nil
               | cons t ts => goTuple t ts
               end in
  let goTuple : GHC.Base.String * a ->
                list (GHC.Base.String * a) ->
                GHC.Base.String :=
    fun arg_3__ arg_4__ =>
      match arg_3__, arg_4__ with
      | pair s x, ts => Coq.Init.Datatypes.app s (go ts)
      end in
  go.
```

The need for this strategy shows up in the functions `CoreUtils.stripTicksE` and `CoreUtils.stripTicksT`.[21] These functions recurse through the tuples we find in `Bind` and in the `Case` constructor of `Expr`, and when they recurse, they use auxiliary functions that operate on these tuples parametrically in some of their components. It is these functions, the analogs of `goTuple`, that require the use of **`set type`**.

7.5. Removing coinduction from GHC

As we discussed back in Section 3.2.4, `hs-to-coq` generally assumes that Haskell code is inductive. While we can make individual types coinductive with the **`coinductive`** edit (Section 4.1.7), we chose to work primarily with inductive types for a variety of reasons outlined above. But among the reasons we outline above to focus on inductive types instead of coinductive types – ease of use, programmer practice – one stands out in this context: from the beginning, we had our eyes on verifying pieces

[21] See Section 7.7.4 for more detail on why and how we translate these functions.

of GHC. And when considering our design choices, we considered what we would need to represent an abstract syntax tree. "Programs are finite," we thought, "so surely inductive types are the correct representation to use." And so things continued happily, right until we got into the weeds of translating GHC.

One of the core data types of GHC's AST is `IdInfo`, a type that records *info*rmation – metadata – about *id*entifiers. As we discussed in Section 7.3, this type is part of the big cluster of mutually-recursive modules and types that define the central components of GHC, including the type `Expr` of Core expressions. As we saw, from Coq's perspective, this collection is uncooperative; but the mutuality is not the only way this is so. Among the pieces of information stored by an `IdInfo`, we find two sorts that cause problems. The first is that the types may contain functions which take other types that are part of this same mutually-recursive morass as input; this is a violation of the strict positivity condition Coq requires for all data types. The second problem is more surprising: it turns out that some of the metadata stored by `IdInfo` may be coinductive.

The source of this coinduction lies in optimization information: identifiers have an *unfolding*, which is used during inlining, and may have attached *rewrite rules*, which are applied during optimization (Peyton Jones, Tolmach, and Hoare, 2001). These may be generated from annotations in the source code, or they may be generated automatically by the compiler. (For example, unfoldings can be produced when the compiler decides to inline a small function or when the user adds an `{-# INLINE #-}` pragma; rewrite rules are used automatically to specialize class methods or written by the user in a `{-# RULES #-}` pragma.) These attached pieces of information may themselves reference other identifiers; because Haskell is lazy, these references are to full-fledged identifiers, which may themselves have these annotations, *ad infinitum*.

So what do we do here? Since `Expr` and `IdInfo` (and the other types; I'll refer to them all synecdochally as `Expr`) are coinductive, and coinduction produces larger types that contain their inductive equivalents, we could just make `Expr` and friends coinductive with `coinductive` edits. However, doing this would be horrible. Firstly, coinduction in Coq is just more challenging to work with in practice, and we would lose access to all the standard inductive and recursive techniques. But more importantly, GHC *also* treats `Expr` as inductive. Recall that the coinduction is localized to the metadata in `IdInfo`; other parts of the data structure are treated inductively. For instance, consider the `exprSize` function, shown in Figure 7.4, that computes the size of a Core expression. This is a classic structurally-recursive function, and works because, just as we thought when we started, programs are finite objects. In other words, `Expr` and friends are neither inductive nor coinductive, but *both*, depending on which portion of the recursion is being considered. I have discussed how types like `[]` are used both inductively and coinductively at different use sites (back in Section 3.2.4); this is a similar problem but for a *single* group of mutually recursive data types.

The solution – to both this coinductive problem and the strict positivity problem mentioned above – is to use edits to elide the problematic details of this mutually-recursive tangle of types. Thankfully, since the non–strictly-positive and coinductive components are only in metadata that isn't relevant to our verification goals, this does

not cause any further problems. We use the `redefine` and `skip constructor` edits to slice out the portions of the data types that caused problems:

```
redefine Inductive  Core.RuleInfo : Type := Core.EmptyRuleInfo.

skip constructor Core.BootUnfolding
skip constructor Core.OtherCon
skip constructor Core.DFunUnfolding
skip constructor Core.CoreUnfolding
```

The original definition of `RuleInfo` contains two fields, one for the rules and one for the free variables:

```
data RuleInfo
  = RuleInfo
        [CoreRule]
        DVarSet
```

The first edit reduces it to a type with zero fields instead. The latter four edits eliminate four of `Unfolding`'s five constructors, reducing the type to simply

```
Inductive Unfolding : Type := | NoUnfolding : Unfolding.
```

These edits arise because we are working with a version of Core that contains no metadata, types, or coercions, and where rule information and unfoldings are metadata. This lack is an invariant we maintain of our translated data structure.

The `skip constructor` edits also eliminate matches on the skipped constructors, which is simpler than the situation for `RuleInfo`. Since the new definition of `RuleInfo` has a single brand new constructor, we need to redefine values that interact with `RuleInfo` as well, such as by

```
redefine Definition Core.emptyRuleInfo   := Core.EmptyRuleInfo.
redefine Definition Core.isEmptyRuleInfo  : Core.RuleInfo -> bool
   := fun x => true.
redefine Definition Core.ruleInfoFreeVars : Core.RuleInfo -> ↩
                                            Core.DVarSet
   := fun x => Core.emptyDVarSet.
```

(Note that the `redefine` edit is one of the few edits that can span multiple lines; all of the line breaks above except for the single marked one are present in the real source code, and not inserted due to page width constraints.) Along the way, these edits have helped break up the mutually-recursive types into smaller groups, simplifying our proof work later on.

We could instead have chosen to replace these types with axiomatized versions. However, this comes with downsides. First, the fact that we can keep an existing constructor of `Unfolding` is "nicer", as it provides more of a connection to the original, and has the potential to let us reuse slightly more code. A bigger advantage, however, is that both of these approaches produce types with true inhabitants. This means that we don't need to axiomatize up values of these types if we need to store them in

types or pass them in functions, and has the potential to make computation easier as it can proceed by evaluation.

These simplifications provide what we need out of `RuleInfo` and `Unfolding`, but of course, the corecursion is in fact used. Functions that operate on Core often need to update the metadata, and this needs to be adjusted. For example, the definition of substitution calls the function `CoreSubst.substRecBndrs` when substituting in a mutually-recursive group (a fitting example):

```
-- | Substitute in a mutually recursive group of 'Id's
substRecBndrs :: Subst -> [Id] -> (Subst, [Id])
substRecBndrs subst bndrs
  = (new_subst, new_bndrs)
  where -- Here's the reason we need to pass rec_subst to subst_id
    (new_subst, new_bndrs) =
      mapAccumL (substIdBndr (text "rec-bndr") new_subst)
                subst
                bndrs
```

(This code was taken from GHC and reformatted; the comments are in the original.) Here, we see that the substitution "ties the knot" with `new_subst`, which looks like a problem. But it turns out that `substIdBndr` only uses that substitution to update the metadata – and not merely metadata in general, but the coinductive unfoldings and rules that we have already eliminated. Thus, we can safely replace `new_subst` with any well-typed term, as it will never be used by our translation. We enact this with the edit

```
in CoreSubst.substRecBndrs ↩
  rewrite forall x, ↩
    CoreSubst.substIdBndr x new_subst = ↩
    CoreSubst.substIdBndr x (GHC.Err.error Panic.someSDoc)
```

to replace the use of `new_subst` with a call to `error`. Applying this edit results in a function without any corecursion – or recursion – at all (reformatted):

```
Definition substRecBndrs
    : Subst -> list Id -> (Subst * list Id)%type :=
  fun subst bndrs =>
    let 'pair new_subst new_bndrs :=
          mapAccumL
            (substIdBndr
              (Datatypes.id (GHC.Base.hs_string__ "rec-bndr"))
              (GHC.Err.error Panic.someSDoc))
            subst
            bndrs in
    pair new_subst new_bndrs.
```

While this rewrite may seem destructive, we know that it is safe because we have already eliminated the relevant metadata; what's more, if we accidentally made a mistake, our proof will encounter the call to `error`, blocking us from continuing and prompting us to go back and fix the problem.

7.6. Axioms vs. rewrites

As seen above, when verifying parts of GHC, we focused less on generating Coq code with an exact correspondence to the Haskell than we had in the past. This meant we were more willing to elide details of the implementation, and *this* meant we had to make some design decisions about how to do so. One particularly interesting decision had to do with the representation of types and coercions. Since we were only interested in verifying term-level operations that don't rely on type information (see the next section for more details), we do not need to analyze the types `Type` and `Coercion` of types and coercions (as mentioned in Section 7.4). However, even if we don't analyze them, we can't simply reduce them to unit or empty types, since the code does assume there *are* many distinct types and coercions. This means that we cannot simply use **`redefine`** or **`skip constructor`** as we did for `RuleInfo` and `Unfolding` in the previous section. Instead, we axiomatize these types. We define the Coq axioms

```
Axiom Coercion          : Type.
Axiom Type_             : Type.
```

in the handwritten module `AxiomatizedTypes`, and then attach the existing types to them with the edits

```
rename type TyCoRep.Coercion      = AxiomatizedTypes.Coercion
rename type TyCoRep.Type_         = AxiomatizedTypes.Type_
```

Both `Type` and `Coercion` are defined in the module `TyCoRep`; the type `Type` in that module was automatically renamed to `Type_` because `Type` is a reserved word in Coq. Additionally, when translating `TyCoRep`, we also suppress the translation of those types:

```
skip AxiomatizedTypes.Coercion
skip AxiomatizedTypes.Type_
```

The final result is that all references to the original `Type` and `Coercion` are replaced by references to the new axiomatized versions, and their definitions are skipped. These edits also have the pleasant side effect of reducing the size of the set of mutually-recursive types, as the original `Type` and `Coercion` are very involved in the interdependence. The reason we used the handwritten module `AxiomatizedTypes` is to help break the mutual recursion between the modules and types that make up Core; since the types are now axioms, they have no dependencies, and since we've manually placed them all in one module, that module can now come earlier in the dependency chain.

However, now that we have done this, we have further problems: all the functions that match on or directly construct types and coercions no longer work. And we can't

actually give those functions meaningful definitions, since we're axiomatizing the types they depend on. So how can we best handle this?

To keep things concrete, let's consider the following two functions from `TyCoRep`, which compute the free type variables in a `Type` and a `Coercion`, respectively:

```
tyCoFVsOfType :: Type -> FV
tyCoFVsOfCo :: Coercion -> FV
```

Thanks to axiomatizing `Type` and `Coercion`, we can treat these functions as always returning empty sets of free variables. However, we don't get that for free; instead, since both these functions are defined by pattern matching, they need to be handled directly in order for the translations to be usable.

The simplest approach is to axiomatize these functions with edits, and then axiomatize their behavior in the specification. We could accomplish this by adding the edits

```
axiomatize definition Core.tyCoFVsOfType
axiomatize definition Core.tyCoFVsOfCo
```

which would generate the Coq definitions

```
Axiom tyCoFVsOfType : Core.Type_ -> Core.FV.
Axiom tyCoFVsOfCo : Core.Coercion -> Core.FV.
```

Then, when writing our specification, we could add axioms specifying their behavior:

```
Axiom tyCoFVsOfType_empty : ↩
  forall ty, Core.tyCoFVsOfType ty = FV.emptyFV.
Axiom tyCoFVsOfCo_empty   : ↩
  forall co, Core.tyCoFVsOfCo   co = FV.emptyFV.
```

Anywhere in our proofs that we got stuck on an application of `tyCoFVsOfType` or `tyCoFVsOfCo`, we could then rewrite by the corresponding `_empty` axiom to proceed.

However, this is not what we do. Instead, we take a different approach: since we know that applying these functions should always produces `FV.emptyFV`, why not encode that with a rewrite rule instead? This means that we don't need the original functions at all, so we have the following four edits:[22]

```
rewrite forall ty, Core.tyCoFVsOfType ty = FV.emptyFV
rewrite forall co, Core.tyCoFVsOfCo   co = FV.emptyFV

skip Core.tyCoFVsOfType
skip Core.tyCoFVsOfCo
```

We now no longer need the `_empty` axioms, and in fact have no axioms at all. All calls to the functions are replaced by `FV.emptyFV`, and the `tyCoFVsOfType` and `tyCoFVsOfCo` functions no longer show up in the translated code.

Why do we take this approach rather than the perhaps more direct axiom approach? There are two main reasons:

[22] I've adjusted the spacing, but these are all applied in our actual development

(1) It makes our proofs easier.
(2) It is guaranteed to be sound.

The first reason, namely that using `rewrite` makes our proofs easier, comes about because `hs-to-coq` is doing more work for us. In the axiom variant, we have to use Ltac to `rewrite` by an axiom (that we defined!) whenever we need these functions to compute during a proof. By using the `rewrite` edit, on the other hand, this computation is done during translation ("at compile time", if you will), and there is no need for us to think about performing it ourselves.

The second reason, namely that `rewrite` edits are guaranteed to be sound but `Axiom`s are not, is a little more interesting. These two approaches seem like they ought to be the same – both allow a function to compute to a specific value in all cases. However, `Axiom`s can be inconsistent. There would be nothing stopping us from adding both the `_empty` axioms as well as

```
Axiom tyCoFVsOfType_unit :
  forall ty, Core.tyCoFVsOfType ty = FV.unitFV someVariable
```

Now, we can prove that `FV.emptyFV = FV.unitFV someVariable`, and thus probably derive a contradiction. On the other hand, `rewrite` edits can only replace one Gallina term with another; the resulting code is type checked, and if this succeeds, then we know that we did not introduce any inconsistency.

This approach does have downsides; in particular, `rewrite` edits leave no trace in the generated Coq code. If a `rewrite` edit has gone awry – say, if we rewrote `Core.tyCoFVsOfType ty` to `FV.unitFV someVariable` – the user may not know why their code is filled with arbitrary terms that they don't understand. We can also use `Print Assumptions` in Coq to check what axioms a term depends on; `rewrite` edits apply silently, and cannot be tracked in this way.

The tradeoffs between these two approaches are real, and we settled on the `rewrite` edit approach because its benefits exceeded the drawbacks for our work on GHC. But this is another iteration of the observation that when we relax our desires for input-output similarity (from "exact" to "as good as we can while completing the proofs"), there are more design choices that need to be made, and more possible solutions to those designs. I have had to ensure that `hs-to-coq` and the edit language remain powerful enough to express these different techniques, because we can no longer assume that one size fits all; as we can see here, the different pieces of the edit language can fit together to give rise to these multiple solutions.

7.7. Justifying edits with proofs

Most of the previous discussion of edits in this chapter has shown us how to use them to make dramatic changes to the Haskell code; this is in contrast to the previous chapters, which use edits to adjust things but largely preserve both the semantics and most of the structure of the Haskell code through the translation. A third kind of situation that shows up in GHC is a combination of the two: we might need to perform a more dramatic transformation of the code between two equivalent representations, if Coq can more easily process the new version. For instance, this commonly comes

up when we need to restructure code so that it will be accepted by Coq's termination checker.

We can, of course, use an edit to perform these transformations, but this does not guarantee the equivalence that we desire. While working on GHC, I began to develop a technique for increasing our confidence in these edits: we could prove that they were correct in Coq, and then annotate the edits with comments appealing to these theorems. While this connection is not live, it provides some justifications that these edits are correct, even though we can't verify them. This technique is already giving us increased confidence, but its final form is not yet settled; we are still exploring the design space of coupling Coq proofs with `hs-to-coq` edits.

In this section, I first demonstrate why we might want to rewrite an expression to equivalent code, using a real example from GHC (Section 7.7.1). I then discuss a fine point of Coq's termination checker which drives most of our justified edits (Section 7.7.2). Finally, I discuss the remaining justified edits that we use in our translation of GHC, both those that are there because of the termination checker and those that are not (Sections 7.7.3 and 7.7.4). This latter category includes an edit with a bug introduced because the justification of edits is not live, providing a look at the tradeoffs of this developing technique.

7.7.1. Why rewrite to equivalent code? The first question one might ask, when confronted with this technique, is: "why bother using an edit to transform between two equivalent representations?" After all, if the two representations are equivalent, it seems like this would be unnecessary. As I alluded to above, the most common answer to this question is: "to satisfy Coq's termination checker." This is not our only strategy to deal with termination: as we have discussed previously, we can use the **`termination`** edit to alter the translation of functions to use more sophisticated termination arguments than Coq supports as long as we provide a proof. However, sometimes this termination proof may be complicated, and the function in question may be almost structurally inductive *if only* Coq could see a little further. In these cases, an alternative approach is to use a **`rewrite`** (or similar) edit to change the offending part of the code to make it more transparent to Coq.

One problem of this nature we ran into was the following recursion pattern. Recall the mutually recursive types `Expr` and `Bind` of Core expressions and binders from Figure 7.2. In its `Rec` constructor, the type `Bind` contains a list of `Expr`s; thus, functions on `Bind` may be mutually recursive via calls to higher-order functions like `map`, or more generally via any recursive function on lists. However, if not handled carefully, Coq can quickly become unable to see through this sort of recursion; even a little complexity can cause a problem.

Consider, for instance, the Haskell functions `freeVars` and `freeVarsBind` in the module `CoreFVs`. These functions annotate `Expr`s and `Bind`s, respectively, with their sets of free variables "at every tree node" as per a comment in the source; this means that, for instance, a value `App e1 e2` would have `e1` annotated with its free variables, `e2` annotated with *its* free variables, and the whole `App` node annotated with the union of those two sets. These functions are also mutually recursive. The details of how they calculate these free variable sets are not important here; instead, what we care

about is certain details of the implementation of `freeVarsBind`. Those details are in the `Rec` case of the definition, which we present here, dimming out the portions we don't care about:

```
freeVarsBind (Rec binds) body_fvs
  = ( AnnRec (binders `zip` rhss2)
    , delBindersFV binders all_fvs )
  where
    (binders, rhss) = unzip binds
    rhss2           = map freeVars rhss
    {- .. omitted ... -}
```

We start with the argument to the function, `Rec binds`, whose contents are a list of pairs of variable binders and their right-hand sides. This list is then taken apart with the function `unzip :: [(a,b)] -> ([a],[b])`, producing two parallel lists: one list of the binders, and one list, `rhss`, of their right-hand sides. The function then recurses by mapping `freeVars` over `rhss`.

This function always terminates: `unzip` does not create new data, but simply restructures the contents of its arguments, and `freeVarsBind` recurses on the output of `unzip`. But if we translate this function to Coq, then Coq cannot see this: since Coq's termination checker is strictly syntactic, the fact that `rcqs` *semantically* contains only subterms of `binds` does not matter. Thus, in order to get Coq's termination checker to accept `freeVarsBind`, we need to ensure that the recursive calls are directly on `binds`. Luckily, this is something we can do: `unzip binds` is identical to `(map fst binds, map snd binds)`, so we could rewrite the recursive call as `map freeVars (map snd binds)`, and then fuse the two maps together as `map (freeVars ∘ snd) binds`. Because we are now recursing on `binds` directly, Coq can see that this call to `freeVars` is indeed on a subterm of the original input (`Rec binds`), and will verify that `freeVarsBind` is terminating.

This is the exact change I applied to this code with `hs-to-coq` for our translation of GHC, providing the following edit and commentary:

```
# Justified by
#   • In `let 'pair _ ys := GHC.List.unzip xys in ...`, we know
#     that `ys = GHC.Base.map snd xys`
#     (`Proofs.GHC.List.snd_unzip`).
#   • Successive `map`s can be converted to one `map` plus function
#     composition (`Proofs.GHC.Base.map_map`).
in CoreFVs.freeVarsBind rewrite forall, ↩
  GHC.Base.map CoreFVs.freeVars rhss = ↩
  GHC.Base.map (CoreFVs.freeVars GHC.Base.∘ snd) binds
```

I include the comment here because comments like it are a key part of this new approach of justifying edits. The previous paragraph of text explained why this transformation was safe, and the comment here does the same more tersely and via an appeal to two

Coq theorems: `Proofs.GHC.List.snd_unzip` and `Proofs.GHC.Base.map_map`. These theorems are from our specification of the `base` library:[23]

```
Lemma snd_unzip:
  forall a b (xs : list (a * b)),
  snd (List.unzip xs) = map snd xs.
Proof.
  intros.
  induction xs.
  * reflexivity.
  * simpl. repeat expand_pairs. simpl. f_equal. apply IHxs.
Qed.

Lemma map_map {a b c} (f : a -> b) (g : b -> c) (x : list a) :
  map g (map f x) = map (g ∘ f) x.
Proof. by rewrite !hs_coq_map Coq.Lists.List.map_map. Qed.
```

So long as the prose description of how they apply to the **`rewrite`** edit is correct, these two theorems *prove* that the transformation done by the **`rewrite`** is correct and so does not change the behavior of `freeVarsBind`. By adding the comment pointing to these Coq proofs, we have significantly increased our confidence that the associated edit is correct.

7.7.2. Recursion through nested fixpoints. When translating `freeVarsBind`, we were able to appeal to general lemmas about general-purpose functions from `base`, but in general, we found that we needed to provide custom reasoning about translations involving custom functions. In order to understand why we needed to provide those translations while translating GHC, we must first pause here to understand why Coq was able to see through the call to `map` above, when it wasn't able to see through the call to `unzip`. Understanding when Coq can verify recursion through nested fixpoints like this is key to understanding why we needed to make most of the edits that we justified.

We saw above that Coq was able to correctly understand that the expression `map (freeVars ∘ snd) binds` only contained structurally recursive calls to `freeVars`, but that it was not able to understand the same of (the equivalent of) the expression `map freeVars (snd (unzip binds))`; indeed, it also would not have understood `map freeVars (map snd binds)`, because Coq never propagates informational about structural subterms through function calls. It is the fact that `freeVars` and `binds` both occurred directly in the argument to a single call to `map` that mattered. But what makes `map` special? Why does this work?

The answer has to do with how Coq's termination checker interacts with nested fixpoints: just *why* Coq can see through the use of `map` above and, more importantly, when it can't (Herbelin, 2010). The situation where this issue arises is one where we have a *nested inductive type*: an inductive type which has a recursive occurrence within another recursive type. In order to write recursive functions on such types, we

[23] As you can see, the collaborative nature of this project gives rise to a variety of proof styles!

correspondingly need *nested fixpoints*: a fixpoint within a fixpoint. This is what we saw above: on the type side, `Bind` is a nested recursive type, as one of its constructors contains a *list* of `Exprs`; and on the fixpoint side, `freeVarsBind` is a fixpoint and it calls `freeVars` recursively through `map`, which is itself implemented in terms of a fixpoint.

To understand Coq's behavior around this, we'll look at something simpler than Core's `Bind` type: the type of nonempty rose trees. These trees contain a value and a list of zero or more child trees, which makes them about as simple as nested inductive types come:

```
Inductive Tree a :=
  Node : a -> list (Tree a) -> Tree a.
Arguments Node {_} _ _.
```

Now that we have a simple nested inductive type, we also need a simple function on it. We'll define the map function on rose trees, which applies a function to every value in the tree; we call this function `map_tree`. In order to define this, we have to recurse over the list of child trees; the natural way to do that is to map over said list. We can thus now give the definition of `map_tree`, highlighting the call to `map` which performs the nested recursion:

```
Fixpoint map_tree {a b} (f : a -> b) (t : Tree a) : Tree b :=
  let 'Node x ts := t
  in Node (f x) (map (map_tree f) ts).
```

As we saw above, Coq can see through this definition when we use the standard library's `map` (which we use as the body of our translation of the Haskell `map` from `base` via a `redefine` edit). But what is that definition?

The natural definition of `map` in Coq is the following, which we call `map1`; we use the name `g` for the function argument so we can refer to it separately from the function argument to `map_tree`.

```
Fixpoint map1 {a b} (g : a -> b) (xs : list a) : list b :=
  match xs with
  | nil       => nil
  | cons x xs => cons (g x) (map1 g xs)
  end.
```

This defines a single fixpoint that takes four arguments (two types, a function, and a list) and produces a list. Unfortunately, using this for the `map` in `map_tree` fails: Coq reports that the "[r]ecursive call to `map_tree` has not enough arguments" if we specify that `map_tree` recurses via `t` with a `{struct t}` annotation; without that annotation, Coq is sufficiently confused that it simply says that it "[c]annot guess decreasing argument of `fix`."

The definition of `map` that we need to use instead is the following, which we call `map2`; it is extensionally equivalent to `map1`, but not identical.[24]

```
Definition map2 {a b} (g : a -> b) : list a -> list b :=
  fix map2_rec xs :=
    match xs with
    | nil       => nil
    | cons x xs => cons (g x) (map2_rec xs)
    end.
```

Here, the first three arguments to `map2` are provided through a (non-recursive) function; the result is a fixpoint of *one* argument which closes over those parameters and actually performs the recursion underlying `map`. When we use this definition for the `map` in `map_tree`, Coq is happy.

To see the difference between `map1` and `map2`, consider what happens when we β-reduce the body of `map_tree`. In the latter case, since `map2` is a function that returns a fixpoint, the definition of `map_tree` becomes

```
Fixpoint map_tree {a b} (f : a -> b) (t : Tree a) : Tree b :=
  let 'Node x ts := t
  in Node (f x) ((fix map2_rec xs :=
                    match xs with
                    | nil       => nil
                    | cons x xs => cons (map_tree f x) (map2_rec xs)
                    end) ts).
```

Within `map2_rec`, its `g` parameter is replaced with `map_tree f x`. Coq can now see that the recursive call to `map_tree` is on a subterm of `t` as through transitivity: first, `map_tree` is applied to `x`; second, `x` is a direct subterm of `xs`, the argument to `map2_rec`; third, `map2_rec` is applied to `ts`; and fourth, `ts` is a direct subterm of `t`. The tricky piece of reasoning is the passage from the second observation to the third: Coq *is* willing to propagate subterm information down through the inputs to fixpoints, so within this call to `map2_rec`, every subterm of `xs` is known to be a subterm of `ts`. This is exactly the nested generalization of the condition for nonnested fixpoints: just as Coq can see that the nested occurrence of the type `Tree a` within `list` is permissible, it can see that the nested call to the fixpoint on `Tree` within the fixpoint on `list` is permissible. Coq does not go any further than this; in particular, Coq does not propagate subterm information on to function *outputs*, either in the nested or nonnested case, which is why the versions of `freeVarsBind` with `unzip` or with two calls to `map` were not accepted by the termination checker.

Contrast this situation with what would happen if we used `map1`: we would wind up with the body of `map_tree` containing

```
Node (f x) ((fix map1 {a b} g xs := ...) (map_tree f) ts)
```

[24]The definition of `map` in the Coq standard library is effectively the same, but written in terms of a Coq `Section` (The Coq Development Team, 2020a). We use that standard `map` as the body of our translation of the Haskell `map` from the `base` library via a `redefine` edit.

and could do no further reduction, since fixpoints don't reduce until applied to a constructor. Because Coq's termination analysis is strictly syntactic and the recursive call to `map_tree` is not being applied to a subterm of `t` (in fact, it's not being applied to anything), Coq cannot accept this definition. For Coq to accept this, it would have to see that the fixpoint could be partially applied, perhaps by seeing that `g` remains unchanged at every recursive call – effectively, it would have to realize that `g` (and `a` and `b`) could be "pulled out" to turn `map1` into `map2`. This is a deeper syntactic analysis than the termination checker will perform, so we must use `map2` directly instead.

What we have seen here is that, in order to aid Coq's termination checker, it is critical to ensure that function arguments to fixpoints are closed over instead of passed directly if there is any chance that the higher-order fixpoint in question will ever be used in a nested context. In this example, that means using `map2` instead of `map1`. Furthermore, even if that's been done, the higher-order fixpoint containing a recursive call must be applied *directly* to a subterm of the input. If there are one or more intervening fixpoints between the one with the recursive call and the subterm, they must somehow all be "fused" into one fixpoint containing the recursive call. In the case of `freeVarsBind`, this is the difference between `map (freeVars ∘ snd) binds`, which works, and `map freeVars (map snd binds)`, which does not.

Unfortunately, these conditions don't come naturally when working with `hs-to-coq`. The output it generates is of the natural, `map1` form, which means we cannot satisfy the first requirement; and Haskell code is often written as the composition of many smaller functions, which means we often do not satisfy the second requirement. We have already seen the first edit we apply to deal with this problem: we **`redefine`** the `map` function from `base` to be implemented in terms of Coq's `map`, which smooths the recursion situation over in many places. More generally, in order to avoid both of these problems and allow Coq to verify that our nested recursive functions are structurally terminating, we need to rewrite some subterms of recursive functions to refer to *new, custom fixpoints* that are written to satisfy these requirements. In the case of `freeVarsBind`, instead of a new fixpoint, we could just reuse `map`, but in general, we need more specific functionality. And because we need this specific functionality but don't want to change the meaning of the code, we want to justify these edits by pointing to Coq equality proofs.

7.7.3. Justified edits to satisfy the termination checker. All told, there are eight places in our GHC translation that applied this technique of using justified edits. Six of these cases, including the `CoreFVs.freeVarsBind` case we discussed in Section 7.7.1, were for edits that enabled Coq's termination checker to understand recursion through nested fixpoints; the remaining two cases were about removed code, and I discuss them in the next section.

As previously mentioned, each of the five nested fixpoint cases besides `freeVarsBind` required writing a custom function. In order to organize our code, we keep all these functions and the custom correctness proofs in a single Coq module called `NestedRecursionHelpers`. The simplest use of that module is the second edit we apply when translating `CoreFVs`: a simple unscoped **`rename value`** edit to

replace a function's uses wholesale. This is the only justified edit added by one of my collaborators, namely my advisor Stephanie Weirich; it is the first example of this technique being adopted by another user of `hs-to-coq`.

```
rename value ↩
  Util.mapAndUnzip = NestedRecursionHelpers.mapAndUnzipFix
```

The function `mapAndUnzip` has type `(a -> (b, c)) -> [a] -> ([b], [c])`, and is equivalent to the composition of `unzip` and `map`; however, it is implemented in GHC as a single recursive function. This is already convenient for us, since this sort of fusion of recursive functions is one of the components of satisfying the termination checker with nested fixpoints. It isn't enough, though, because, as discussed above, `hs-to-coq`'s translation of `mapAndUnzip` passes all the arguments, including the function, to the resulting `fix`point. Thus, we define an almost-identical function, `NestedRecursionHelpers.mapAndUnzipFix`, that simply keeps the first argument, the function, fixed between recursive calls; the `map2`-style implementation to `hs-to-coq`'s default `map1`-style implementation, using the names of the examples we saw in the previous section. Renaming the original function to the replacement then allows us to translate `CoreFVs.freeVars`, which uses `mapAndUnzip` when computing the free variables of different arms of a `Case`. (The edit would also alter any other uses of `mapAndUnzip` in `CoreFVs`, but this was the only one.)

Normally, if we were doing such a global renaming, we could then skip `Util.mapAndUnzip`; however, Coq has no trouble with the definition of `Util.mapAndUnzip`, just its nested use in `CoreFVs.freeVars`. Thus, we translate `Util.mapAndUnzip` and include a proof (also in `NestedRecursionHelpers`) that our new definition is the same as the old one:

```
Theorem mapAndUnzipFix_is_mapAndUnzip
    : forall a b c (f : a -> (b * c)) l,
  mapAndUnzip f l = mapAndUnzipFix f l.
```

We note that this edit is justified in a comment, although the comment doesn't give the specific theorem name:

```
# Make it easier for Coq to see termination
```

The next two justified edits we consider are the ones we apply in the module `CoreUtils`, both of which I added in order to enable Coq to accept the translation of the function `eqExpr`. This function compares two Core expressions for α-equivalence, and has the type `InScopeSet -> CoreExpr -> CoreExpr -> Bool`; we translate this function, but do not verify it, as none of our final theorems (Section 7.8) require it. It is defined in terms of a recursive function on `Expr` (not mutually recursive with `Bind`, interestingly enough[25]); since this inner function compares two terms for α-equivalence, it needs to compute the conjunction of multiple recursive calls in the `Let (Rec ps)` and `Case` cases, confirming that all the bindings or case arms, respectively, are also α-equivalent. This is done through the function `Util.all2`, which is defined as follows:

[25]The Haskell code is mutually recursive, but inessentially – this mutual recursion is eliminated in the translation via an `inline mutual` edit.

```
all2 :: (a -> b -> Bool) -> [a] -> [b] -> Bool
-- True if the lists are the same length, and
-- all corresponding elements satisfy the predicate
all2 _ []     []     = True
all2 p (x:xs) (y:ys) = p x y && all2 p xs ys
all2 _ _      _      = False
```

As the comment says, this function returns `True` if and only if the two lists have the same length and the supplied predicate returns `True` for each pair of elements at the same index. Put another way, it is equivalent to the composition of `all` and `zip`, with the additional requirement that the lists be the same length.

As we have seen by now, `all2` is translated by `hs-to-coq` in such a way that `p` is bound by the resulting `fix`point. However, this time, just swapping `all2` for a definition that closes over `p` is insufficient, because of the same problem we had back in Section 7.7.1 with `CoreFVs.freeVarsBind`: one of the recursive calls is actually applied to `unzip`. I thus needed to fuse `all2` with `map snd`, much as we saw before. I could have done this via a replacement definition of `NestedRecursionHelpers.all2`, followed by bringing `snd` in directly by rewriting the offending application from `Util.all2 p` to `Util.all2 (fun x y => p (snd x) (snd y))`; however, I decided that it would be clearer to instead defined a single fused function `NestedRecursionHelpers.all2Map`, which takes two transformation functions and applies them to the respective input lists before forwarding the result to the predicate. I also proved that this function is equivalent to the composition of `Util.all2` with two `map`s, which is the theorem that justifies the edits where we replace `Util.all2`.

```
(* `all2Map p f g xs ys = all2 p (map f xs) (map g ys)` (see
   `all2Map_is_all2_map_map`).

   We need this for use in `CoreUtils.eqExpr` so Coq can see that
   it's terminating.  We also need to replace `unzip` with
   `map snd`, which we don't justify here.

   [...]  *)
Definition all2Map {a b a' b'}
                   (p : a -> b -> bool)
                   (f : a' -> a) (g : b' -> b)
    : list a' -> list b' -> bool :=
  fix all2Map xs0 ys0 :=
    match xs0 , ys0 with
    | nil     , nil     => true
    | x :: xs , y :: ys => p (f x) (g y) && all2Map xs ys
    | _       , _       => false
    end.

Theorem all2Map_is_all2_map_map {a b a' b'}
```

```
                                    (p : a -> b -> bool)
                                    (f : a' -> a) (g : b' -> b)
                                    xs ys :
  all2Map p f g xs ys = all2 p (map f xs) (map g ys).
Proof.
  elim: xs ys => [|x xs IH] [|y ys] //=.
  by rewrite IH.
Qed.
```

The comment at the start, though not part of the Coq code, is part of our justification scheme: it points to both the theorem that I proved about it and the edits in which I used it. (The omitted text contains a pointer to an explanation of the problem we're solving; that is, it contains a pointer to a shorter version of Section 7.7.2.)

This proof then backs up the two justified edits we need to apply to `CoreUtils.eqExpr`. The definition of `exExpr` is in terms of a local recursive function called `go`; the relevant portions of `go` are as follows, dimming out the portions we don't care about:

```
go env (Let (Rec ps1) e1) (Let (Rec ps2) e2)
  = equalLength ps1 ps2
  && all2 (go env') rs1 rs2 && go env' e1 e2
  where
    (bs1,rs1) = unzip ps1
    (bs2,rs2) = unzip ps2
    env' = rnBndrs2 env bs1 bs2

go env (Case e1 b1 t1 a1) (Case e2 b2 t2 a2)
  | null a1   -- See Note [Empty case alternatives] in TrieMap
  = null a2 && go env e1 e2 && eqTypeX env t1 t2
  | otherwise
  =  go env e1 e2 && all2 (go_alt (rnBndr2 env b1 b2)) a1 a2

{- ... omitted ... -}

go_alt env (c1, bs1, e1) (c2, bs2, e2)
  = c1 == c2 && go (rnBndrs2 env bs1 bs2) e1 e2
```

There's a bit of noise here, but we can see that in the first case, we're using `all2` to confirm that all the right-hand sides of two recursive let bindings are the same; we can also see that in the second case, we're using `all2` to confirm that the result of every pattern match in a case expression is the same via the local function `go_alt`. (Recall that a `Case` contains a list of `Alt`s, which are triples of the constructor, the binders, and the right-hand side, as we saw in Figure 7.2.)

These two different calls need two different edits. In the first case, we also need to avoid an `unzip`, so we pass `snd` to `all2Map`; in the second case, we just need the basic

functionality of `all2` since `go_alt` is doing the unpacking, so we pass `id` to `allMap2` (in this case, simply using a more cooperative definition of `all2` would have sufficed). The two edits that implement this are as follows:

```
# Justified by `NestedRecursionHelpers.all2Map_is_all2_map_map`
# plus changing
#
#     let '(xs,ys) := unzip xys in ... ys ...
#
# to
#
#     ... (map snd xys) ...
in CoreUtils.eqExpr rewrite forall p, ↩
  Util.all2 p rs1 rs2 = ↩
  NestedRecursionHelpers.all2Map p snd snd ps1 ps2

# Justified by `NestedRecursionHelpers.all2` plus `map id =1 id`.
# Could also work with a better version of `Util.all2`; see
# issue #109.
in CoreUtils.eqExpr rewrite forall p, ↩
  Util.all2 p a1 a2 = NestedRecursionHelpers.all2Map p id id a1 a2
```

The edits distinguish between the two cases by matching on the lists being iterated over by name (`rs1` and `rs2` vs. `a1` and `a2`). The comments justify the correctness of these edits by pointing to the theorem `NestedRecursionHelpers.all2Map_is_all2_map_map`, which we saw above. And indeed, the **`rewrite`**s here look a lot like instantiations of that theorem. (Issue #109 is the GitHub issue[26] that documents that `hs-to-coq`'s default translation runs into the nested fixpoint problem we're addressing here.)

The final two justified edits for termination we consider here are the ones I added to support the translation of *common subexpression elimination* (CSE), a Core-to-Core optimization pass. This pass, which we translate but do not prove, is responsible for taking duplicated expressions in Core programs, binding them to a variable, and then referring to the variable. For example, it could take the two Core variable definitions (written in a Haskell-like syntax)

```
p1 = (x, Just 42)
p2 = (x, Just 42)
```

and replace them with

```
p1 = (x, Just 42)
p2 = p1
```

(CSE is also the sole consumer of `CoreUtils.eqExpr`, the function that we translated with the previous two justified edits, in our translated version of GHC.)

[26] https://github.com/plclub/hs-to-coq/issues/109

This time, unlike in `CoreUtils`, our two edits affect two different portions of CSE, both having to do with how CSE operates on `Binder`s. CSE is defined in the eponymous module `CSE`, and the core of the definition is the pair of mutually-recursive functions

```
cseExpr :: CSEnv -> InExpr -> OutExpr
cseBind :: TopLevelFlag -> CSEnv -> CoreBind -> (CSEnv, CoreBind)
```

These functions perform common subexpression elimination on an expression and a binder, respectively. Here, the types `InExpr` and `OutExpr` are mnemonic synonyms for `CoreExpr`; the type `CSEnv` contains the necessary substitution and maps that allow translating between core expressions and the variables they can be bound to; and the type `TopLevelFlag` tells us whether or not the binder being translated is at the top level (like the definitions of `cseExpr` and `cseBind` themselves) or not (like a `let`-bound variable). These functions are themselves defined in terms of various auxiliary functions, several of which we have to collapse into `cseExpr` and `cseBind` via `inline mutual`. But even once we've done this, there are still two more changes we need to make in the service of termination, both of which we justify.

The first of these changes looks much like what we have seen before. Instead of a combination of `map` and `unzip`, the function `cseBind` uses a combination of `mapAccumL` and `zip`, which does not pass the termination checker for the same reason. The function `mapAccumL` can be thought of in two different ways: as a combination of `map` and `foldl`, as the documentation has it (The Core Libraries Committee, 2018, the `Data.Traversable` module); or as `traverse` (aka `mapM`) in the state monad (modulo the order of function arguments and tuple components), as the implementation has it. The type of `mapAccumL` is given in terms of `Traversable`:

```
mapAccumL :: Traversable t
          => (a -> b -> (a, c)) -> a -> t b -> (a, t c)
```

It applies the provided function to each element of the `Traversable` of `b`s (such as a list of `b`s, `[b]`), passing an accumulator of type `a` along from left to right as well (hence the `L` in the name), and then returning the final value of the accumulator and the results of each function application. For instance, the function

```
enumerate :: Traversable t => t String -> (Int, t String)
enumerate = mapAccumL (\i s -> (i+1, show i ++ ". " ++ s)) 1
```

takes a list or other `Traversable` of `String`s and numbers them, returning a pair of the putative next index and the numbered items:

```
enumerate ["Haskell", "Edits", "Coq"] ==
  ( 4, [ "1. Haskell"
       , "2. Edits"
       , "3. Coq" ] )
```

Just as we have seen before, in order to make the recursion apparent to the termination checker, I combined `mapAccumL` and `zip` into a single Coq function, which I called `NestedRecursionHelpers.zipMapAccumL`, that I also proved to be equivalent to the composition of the original two functions:

```
(* `zipMapAccumL f s xs1 xs2 = mapAccumL f s (zip xs1 xs2)` (see
   `zipMapAccumL_is_mapAccumL_zip`).

   We need this for use in `CSE.cseBind` so Coq can see that it's
   terminating.

   [...]
 *)
Definition zipMapAccumL
    {acc x1 x2 y}
    (f : acc -> (x1 * x2) -> acc * y)
      : acc -> list x1 -> list x2 -> acc * list y :=
  fix go (s : acc) (xs1 : list x1) (xs2 : list x2)
         {struct xs1} : acc * list y :=
    match xs1 , xs2 with
    | nil        , _          => (s, nil)
    | _          , nil        => (s, nil)
    | x1 :: xs1' , x2 :: xs2' =>
      let: (s',  y)  := f s (x1,x2) in
      let: (s'', ys) := go s' xs1' xs2' in
      (s'', y :: ys)
    end%list.


Theorem zipMapAccumL_is_mapAccumL_zip
    {Acc X1 X2 Y}
    (f : Acc -> (X1 * X2) -> Acc * Y)
    (s : Acc)
    (xs1 : list X1)
    (xs2 : list X2) :
  zipMapAccumL f s xs1 xs2 = mapAccumL f s (zip xs1 xs2).
```

We can see in the definition of `zipMapAccumL` that it contains the inlined definition of bind for the state monad, as promised. The comment above `zipMapAccumL`, is, as we saw with `all2`, part of the justification scheme: again, it points to the correctness theorem and the use case for this function. (The omitted text explains the problem we are solving; in other words, it's a shorter version of Section 7.7.2, and is the text that was pointed to by the comment above `all2`.)

The last justified edit for termination is somewhat different. While still solving the same nested fixpoint problem, the transformation involved is much more dramatic, and the correctness theorem is not a simple equality. The function in question this time is `cse_bind`, one of the aforementioned auxiliary functions that's **`inline mutual`**ed. This function does the core of the work of `cseBind` in the non- and mutually-recursive

cases (the single recursive function case is handled specially). In particular, in the mutually-recursive `Rec pairs` case, `cse_bind` is applied to the `pairs` through a combination of `mapAccumL` and `zip` in order to build and propagate the binding information:[27]

```
cseBind toplevel env (Rec pairs)
  = (env2, Rec pairs')
  where
    (env1, bndrs1) = addRecBinders env (map fst pairs)
    (env2, pairs') = mapAccumL do_one env1 (zip pairs bndrs1)

    do_one env (pr, b1) = cse_bind toplevel env pr b1
```

The purpose of `cse_bind` is to take care of the two pieces of CSE for variable bindings: first, it is supposed to apply CSE on the right-hand side of the binder. Second, after doing so, the resulting new right-hand side will be bound to a variable; `cse_bind` is responsible for remembering that this binding exists so that further occurrences of the resulting right-hand side can be eliminated and replaced with the variable that is bound here. In order to implement this behavior, `core_bind` must keep track of special cases where it must replace its basic functionality with something more complicated. The special case of interest to us here is the following, where we dim the code that isn't involved in the recursion:

```
cse_bind toplevel env (in_id, in_rhs) out_id
  {- ... omittted ... -}
  | Just arity <- isJoinId_maybe in_id
      -- See Note [Don't tryForCSE the RHS of a Join Point]
  = let (params, in_body) = collectNBinders arity in_rhs
        (env', params') = addBinders env params
        out_body = tryForCSE env' in_body
    in (env, (out_id, mkLams params' out_body))
  {- ... omittted ... -}
```

In this case, we have found a *join point*, a special kind of lambda; we explain join points in Section 7.8.2, as they are part of our final theorems about GHC, but all that matters here is that they are literal lambdas with a known minimum number of bound variables (the join arity). Join points cannot be duplicated, and so we should never use the entire join point as a common subexpression to eliminate. Instead, we use `collectNBinders` to split the lambda apart after the first several binders. For example, `collectNBinders 2` applied to a representation of the Core program `\x -> \y -> \z -> x+y+z` would return `([x,y], \z -> x+y+z)`. We then recursively apply CSE to this deeper right-hand side that was under the binders (`in_body` in the definition of `cse_bind`; the function `\z -> x+y+z` in our little example)

[27] One thing to note here is that the *first* component of `pairs` is the variable being bound, which isn't mutually recursive; this is why we aren't worried about the definition of `bndrs1`.

using the auxiliary function `tryForCSE`; this function wraps `cseExpr` and was, like `cse_bind`, also `inline mutual`ed.

There are two problems we have here that we didn't have before. The first is that, while we recurse on its output (`in_body`), `collectNBinders` isn't higher order. This means that, unlike in all the previous cases we saw in this section, we can't simply rearrange things so that the recursive call to `tryForCSE` appears in an argument to `collectNBinders`. The second is that `collectNBinders n e` assumes the presence of `n` binders in `e`; if they are not there, the function crashes. We have not yet had to think about how to justify rewrites in the presence of partiality.

We tackle the first-order problem first: `collectNBinders`, instead of being higher-order, has the simple first-order type `Int -> Expr b -> ([b], Expr b)`.[28] As observed above, the problem here is less like the other cases in this section, and more like the problem with `CoreFVs.freeVarsBind` that we saw in Section 7.7.1: we recurse on the result of a recursive function, rather than inside a higher-order function. As outside observers, we understand that `collectNBinders` always returns a subterm of its input; all it is doing is stripping off leading `Lam` constructors, as we can see by its definition (in the module `CoreSyn`, which becomes part of the merged `Core` module in the Coq output):

```
-- | Strip off exactly N leading lambdas (type or value). Good for
-- use with join points.
collectNBinders          :: Int -> Expr b -> ([b], Expr b)
{- ... other function definitions omitted ... -}
collectNBinders orig_n orig_expr
  = go orig_n [] orig_expr
  where
    go 0 bs expr      = (reverse bs, expr)
    go n bs (Lam b e) = go (n-1) (b:bs) e
    go _ _  _         = pprPanic "collectNBinders" $ int orig_n
```

The eventual recursion in `cse_bind` happens on the second component of the tuple returned by `collectNBinders`. Examining the first two cases of `go`, we can see that this will indeed transitively be a subterm of `orig_expr`; what's more, within `go`, Coq can see that `expr` is a *syntactic* subterm of the original input `orig_expr`. As we saw in when defining `map_tree` via `map2` in Section 7.7.2, because `go` recurses directly on its expression argument, Coq can see at every step that the expression argument is a syntactic subterm of the original input; the only difference with the `map_tree` case is that the subexpressions and the original input are the same type.

However, the catch is that as soon as we return `expr` in the base case and leave `go`, this syntactic subterm information is lost: back in `cse_bind`, Coq can't see that `in_body` is actually a subterm of `in_rhs`. Flipped around, the problem is that the recursion happens *outside* `go`. What if we could recurse directly on the right-hand side of the `go 0` case?

[28] Technically, due to currying, this type is also higher-order, but not meaningfully.

To do that, all we need to do is rewrite `collectNBinders` in *continuation-passing style* (in Coq): if we give it an extra function argument `k`, then the base case can be `k (reverse bs) expr` instead. And just as before, if we make sure to close over `k`, then Coq will inline the definition of `k` in the same way. So, when the recursive call to `tryForCSE` is placed in the continuation, Coq can now see that it is on a subterm of the original input. Another way to think about this solution is that it changes `collectNBinders` from a first-order function into a higher-order function, allowing us to implement a solution much like those we saw in the preceding justified edits where we place the recursive call to `tryForCSE` within the newly-provisioned higher-order argument. We provide this CPSed version as the Coq function `NestedRecursionHelpers.collectNBinders_k`:

```
(* `collectNBinders_k n e (fun bs e' => ...) =
   let '(bs,e') := collectNBinders n e in ...`,
   or both functions panicked (see
   `collectNBinders_k_is_collectNBinders`).

   We need this for use in `CSE.cseBind` so Coq can see that the
   recursive call under join points is terminating. *)
Definition collectNBinders_k `{Default r} {b}
                             (orig_n : nat) (orig_expr : Expr b)
                             (k : list b -> Expr b -> r) :=
  let fix go n bs expr {struct expr} :=
      match n , bs , expr with
      | 0 , _ , _       => k (reverse bs) expr
      | _ , _ , Lam b e => go (n - 1) (cons b bs) e
      | _ , _ , _       => panicStr &"collectNBinders_k" someSDoc
      end
  in go orig_n nil orig_expr.
```

To replace `collectNBinders` with `collectNBinders_k`, I needed to write an edit that was able to capture a continuation of the function after `collectNBinders` – it did not need to be the full continuation, so long as it contained the recursive call. In this case, that continuation was easy to find, as the pattern binding in the **`where`** clause in `cse_bind` that destructures the tuple is translated to a `match` in Coq, and the body of the `match` is exactly the continuation we need since it automatically includes everything that depends on the result of `collectNBinders`.

```
# Justified by
# NestedRecursionHelpers.collectNBinders_k_is_collectNBinders
in CSE.cse_bind ↩
  rewrite forall arity in_rhs params in_body k, ↩
    match Core.collectNBinders arity in_rhs with ↩
    | pair params in_body => k ↩
    end ↩
```

```
  = NestedRecursionHelpers.collectNBinders_k ↩
      arity in_rhs (fun params in_body => k)
```

Now that we have this new edit with an appeal to a named correctness theorem, we need to look at that theorem's statement. But as mentioned above, this theorem statement needs to look a bit different, because `collectNBinders` is partial. In the case where there aren't enough binders to collect, it will crash by calling the function `Outputable.pprPanic`. We talked about the related `Panic.panic` function back in Section 7.4; these functions, and more generally all the machinery from GHC's `Panic` module, are what GHC uses to crash when there are internal errors. For instance, GHC will panic when an an invariant is being violated, such as the invariant that `collectNBinders` is called only when there are enough binders is one of these.

Much as with output, we are not interested in the details of a panic, only whether or not (hopefully not) one happened. We might hope that if we could simplify things so that `collectNBinders` and `collectNBinders_k` raised exactly the same error, then by reflexivity of equality we could prove that the two functions were equal in that case as well; however, this isn't true. Because we represent bottom values ($\bot$) as either opaque constants (as with the `Default` type class, which we saw in Section 3.8) or axioms of inhabited types, any evaluation on them gets completely stuck. This is fundamentally different than $\bot$, which can be passed around and which, when evaluated, produces even more $\bot$s. In our situation, even the slightest rearrangement of function call orders around a partial value can result in completely different terms, as the opaque terms or axioms simply will not budge.

To look at this here, we can compare the two error cases. The error case I wrote in `collectNBinders_k` is, as we can see above,

```
panicStr &"collectNBinders_k" someSDoc
```

The error case in `collectNBinders` looks different in Haskell, but we use edits to significantly simplify the various panic functions and the `SDoc` type that they use for pretty-printing. At the end of this, `hs-to-coq` produces the following error case for `collectNBinders`:

```
Panic.panicStr (GHC.Base.hs_string__ "collectNBinders")
               Panic.someSDoc
```

This differs from the error case in `collectNBinders` only by the substring `"_k"` (and in that it doesn't use the `&` notation). But even if we adjusted our definition of `collectNBinders_k` to drop the substring, it wouldn't help. Consider what happens if we do reach an error: both `collectNBinders` and `collectNBinders_k` will stop with a panic. But because the latter is in continuation-passing style, this isn't the same thing. When `collectNBinders` stops with a panic, the pattern binding of the result to `(params, in_body)` in `cse_bind` gets stuck, and Coq is left with a `match` expression that's matching on something that will never reduce. On the other hand, when `collectNBinders_k` stops with a panic, that's it: the continuation, which corresponds to the body of the `match`, is never evaluated, and the expression is equal to the panic. Again, if panics were true bottom values, then matching on them as

`collectNBinders` does would result in a new panic, but that is not achievable; Coq is fundamentally not a partial language.

Thus, our theorem statement must be weakened from just a simple equality. One candidate theorem is *partial correctness*: that `collectNBinders` is equal to `collectNBinders_k` if the invariant that the input has enough `Lam` constructors is satisfied. But in fact, this sort of partial correctness result is too weak, as it says nothing about the other case. This would allow one function to succeed while the other panicked, which is not what we want. We instead want to say that either the functions are the same, or they both panicked; when proving something about Haskell on paper, the result of evaluating $\bot$ would bubble up out of the function automatically and collapse these two cases, but with our Coq representation we must keep them separate. A property we do preserve from common approaches to Haskell's semantics is that, as we said above, we don't want to distinguish between different kinds of panics, much as Haskell semantics usually have just one $\bot$.

Our approach to capturing "this function panicked" is to define a new proposition `panicked : a -> Prop` that tells us if the value is a call to one of the crashing functions from `Panic`. We add this proposition to the translated `Panic` module using an `add` edit (another edit that, like `redefine`, can span multiple lines):

```
add Panic Inductive Panic.panicked {a} : a -> Prop :=
 | PlainPanic `{(GHC.Err.Default a)} {s}     :
     panicked (Panic.panic s)
 | StrPanic   `{(GHC.Err.Default a)} {s} {d} :
     panicked (Panic.panicStr s d).
```

(There are other functions that can panic; should they ever be necessary for a proof, this type can gain more constructors.) This predicate can only look to see if the expression it is looking at is *exactly* a call to one of these panicking axioms: no nesting, no nothing. But with a little judicious framing, this will be exactly what we need to specify the correctness of the edit that introduces `collectNBinders_k`.

Now that we have addressed our two concerns, I can finally state the final edit justification theorem that I proved (reformatted):

```
Theorem collectNBinders_k_is_collectNBinders
        `{Default r} {b} (orig_n : nat) (orig_expr : Expr b)
        (k : list b -> Expr b -> r) :
  (collectNBinders_k orig_n orig_expr k =
   = let '(out_bs, out_expr) := collectNBinders orig_n orig_expr in
     k out_bs out_expr)
  \/
  (panicked (collectNBinders_k orig_n orig_expr k)
   /\ panicked (collectNBinders orig_n orig_expr) ).
```

We can see the two halves of the theorem statement: *either* `collectNBinders_k` is the same as `collectNBinders`, *or* both functions panicked. We can also see that, thanks to the CPS transformation, the correctness disjunct is not a simple equality of function applications; on the right-hand side of the equality, we match on the

result of `collectNBinders` and use `k` to capture a program context containing the two variables from that match. Relatedly, we can also see that the `panicked` checks only refer to the functions, and even for `collectNBinders` do *not* refer to the surrounding context; this is how we can keep track of whether the functions have panicked without expecting them to automatically propagate panics like they would ⊥s in Haskell.

We do simplify as much as possible, in a variety of ways. Many functions around panicking, like around outputting, refer to the type `Outputable.SDoc` which can be pretty-printed; we replace this type with `String`. We then rewrite everything we can to be as simple as possible: we introduce a special axiom `Panic.someSDoc` via the edit

```
add Panic Axiom Panic.someSDoc : GHC.Base.String.
```

and rewrite every reference to an `SDoc` that we can to `Panic.someSDoc`. We also simplify the various panicking functions, with the primitive ones becoming axioms. Simplification here means "simplest set for us to write down", not minimal in any sense; for instance, we add the new panicking primitive `Panic.panicStr` with the edit

```
add Panic Axiom Panic.panicStr : ↩
  forall {a} `{(GHC.Err.Default a)}, ↩
    GHC.Base.String -> (GHC.Base.String -> a).
```

We also use a **`rename value`** edit in our global-to-GHC edit file to replace the function `Outputable.pprPanic` with this new axiom everywhere:

```
rename value Outputable.pprPanic      = Panic.panicStr
```

7.7.4. Justified `Tick` removal. We have now seen six different justified edits that alter code to work with Coq's termination checker. This context was a natural one for the use of justified edits: the need for replacement code could not be avoided; the justification theorems were (mostly) straightforward; and using `hs-to-coq` means we're working in Coq anyway, so proving one more theorem just feels like the right thing to do. But having come up with this new stratagem of justifying pieces of our translation, I was also interested in exploring other possible applications of it. And the other use case that arose naturally came up while I was translating CSE, and was much like something out of Section 7.6: justifying edits that were present in order to simplify the code. In particular, the edits for the `CSE` module contain two **`rewrite`** edits that each entirely eliminated every use of a function. In Section 7.6, we saw the virtues of this approach vs. using a Coq **`Axiom`**; the difference here is that I could still successfully translate the functions we were rewriting away, which meant that I could justify these edits with Coq **`Lemma`**s rather than need to make any bare assertions.

The obvious question here is: why? If we can translate a function and prove that it's equal to something, and we don't need to appease the termination checker, then why not just use the translation instead of increasing the formalization gap by adding an edit? And the answer to that question is that due to the collaborative nature of this project, work was proceeding in parallel on other parts of GHC; in particular, my collaborator and advisor Stephanie Weirich was working on simplifying the general structure of Core at the same time as I was working on translating CSE. The

simplification work that included the function-eliminating `rewrite`s happened after I had set up most of the translation of CSE; once it did, that selfsame simplification enabled me to prove the justification theorems (instead of the specific preservation theorems I had before) that said that eliminating these functions was just fine.

The next question is: was it worth it? And the answer to that question is: I don't know yet! The pros and cons of justifying edits are still in flux, and while this approach may not have gained us anything here, it was still a useful lesson in the design space of this technique. In particular, `hs-to-coq` is designed to enable gradually increasing verification coverage by allowing for translating more code, reducing the number of simplifications, and so on. The ability to change between a proof-driven approach (with the advantages of greater confidence in our code) and a `rewrite`-driven approach (with the simplicity advantages outlined in Section 7.6) is part and parcel of this, and the fact that we can do both is encouraging.

The two functions in question operate on the "profiling ticks" in a Core `Expr`. Recall from Figure 7.2 that the Haskell definition of `Expr` contains a `Tick` constructor, which wraps an expression with some profiling metadata (a `Tickish Id`). The two functions we are considering are

```
stripTicksE :: (Tickish Id -> Bool) -> Expr b -> Expr b
stripTicksT :: (Tickish Id -> Bool) -> Expr b -> [Tickish Id]
```

These two functions both operate on a subset of these profiling ticks, as picked out by a predicate on their contents. The first function, `stripTicksE`, removes that subset of `Tick`s entirely from the expression; the second, `stripTicksT`, returns the contents of that subset of `Tick`s. (Everything `stripTicksE` removes, `stripTicksT` returns.) For example, `stripTicksE (const True)` would remove all the `Tick`s from an expression, and `stripTicksT (const False)` would always return the empty list.

However, when we recalled the Haskell side from Figure 7.2, we might also recall that, as we said in Section 7.2, we completely eliminate `Tick` on the Coq side. This is the simplification that Stephanie Weirich in parallel with my work on CSE. Since we were not interested in proving any theorems about the compiler with regard to this profiling information, she decided to eliminate the `Tick` constructor entirely, in keeping with our general approach towards simplifying GHC. This had two effects for `stripTicksE` and `stripTicksT`:

(1) They both became very boring when translated to Coq; and
(2) Stephanie Weirich added edits to eliminate calls to them, as she did for `Tick`-related functions in general.

On the first hand, `stripTicksE` and `stripTicksT`'s newfound boringness meant that it was easy to prove that they could be removed: the Coq version of the former takes a predicate and becomes the identity function, and the Coq version of the latter takes a predicate and becomes constantly `nil`. The proofs of these theorems are straightforward inductive arguments, and I carried them out in Coq; the resulting theorems are:

```
Lemma stripTicksE_id {b} p (e : Expr b) :
  stripTicksE p e = e.
```

```
Lemma stripTicksT_nil {b} p (e : Expr b) :
  stripTicksT p e = nil.
```

On the second hand, because she was removing ticks, Stephanie Weirich added the following edits:

```
# remove reasoning about ticks
#
rewrite forall xs e, CoreUtils.stripTicksE xs e = e
rewrite forall xs e, CoreUtils.stripTicksT xs e = e
```

Because they weren't consciously added as part of a justified edits scheme, they don't appeal to `stripTicksE_id` and `stripTicksT_nil` explicitly; however, they still serve as the appropriate justification.

7.7.4.1. *An edit bug.* There's one thing that story leaves out: the second edit above isn't quite right! If we look at it again, we see that it rewrites `stripTicksT xs e` to its second argument, `e` – but this isn't even well-typed! Instead of an `Expr b`, `stripTicksT` returns a `[Tickish Id]`. I proved `stripTicksT_nil`, but this edit would have to appeal to an impossible `stripTicksT_id`.

So why wasn't this caught? It wasn't caught because the edits to remove `Ticks` and related information were very thorough, and also eliminated all references to the list of tick information that `stripTicksT` would have returned. The only uses of `stripTicksT` in `CSE` followed the following pattern:

```
let ticks = stripTicksT tickishFloatable expr1
in ... mkTicks ticks expr2 ...
```

The function `mkTicks` is responsible for adding all the annotations in `ticks` to the given expression; without the `Tick` constructor, it can't do anything, so during the simplification, it was also removed with the edit

```
rewrite forall ts e, CoreUtils.mkTicks ts e = e
```

(This edit is in fact found on the line right after the removal of `stripTicksT`.) This means that the output of `stripTicksT` was no longer actually used, and so the fact that it had the wrong type (not merely the wrong semantics) was completely hidden.

This example demonstrates the weaknesses of this technique of justifying edits. While Coq proofs are always correct, they don't have any intentionality – they aren't automatically *about* anything else. This is why we are so interested in making sure that specifications are two-sided and live (Appel et al., 2017): those concepts connect formal proofs to the object that we actually want to verify. And while `hs-to-coq` is deeply invested in making sure its output is live, these justified edits simply are not. The only connection between a Coq proof and the edit it justifies is at most a comment, which allows the theorem and the edit to drift out of sync with no ability to catch that this is happening.

There are still advantages to justifying edits: while things can go wrong, they can't get any worse than not having the justification in place at all. But this approach is still not fully fleshed out, and the tradeoffs are not fully resolved. In future work,

it might be possible to further explore this approach and build more support for it into `hs-to-coq` in such a way that we could ensure that the connection between the justification and the edit was live; however, it is not yet clear how to make such a connection systematically formal.

7.8. Verifying properties of the compiler

The purpose of all the edits we have discussed was to enable us to verify some properties of GHC. As discussed in Section 7.2, this verification work is applied to the Core intermediate language, where we verified that two different optimization passes use variables correctly. What this means for us is two-fold:

(1) All terms are well-scoped.
(2) All join points are used correctly.

In this section, I explain these in more detail, including defining the notion of a "join point".

We are of course not the only people to have verified compilers, including those for functional languages. For instance, CompCert (Leroy, 2009) is a fully formally verified C compiler written in Coq; and CakeML (Kumar, Myreen, Norrish, and Owens, 2014; Tan, Myreen, Kumar, Fox, Owens, and Norrish, 2016), which we discuss further in Section 9.5, is a functional language with a verified compiler implemented by translation from HOL4. Similarly, `hs-to-coq` is not the only way to use Haskell to apply these verification techniques; we demonstrate how to use LiquidHaskell to verify well-scopedness checking for a simple lambda calculus implemented in Haskell in Section 9.4.5.

7.8.1. Well-scoped Core terms. Core, like any lambda calculus, has variables and bindings, and so we must ensure that all variables are bound. As usual, the devil is in the details. GHC uses an explicitly named representation of variables, so we must ensure that each local variable occurs within a binder. At the same time, we must distinguish between *local* and *global* variables – only local variables must be bound in this way.

Variables in GHC have a rather complicated representation, which we need to use `hs-to-coq` to simplify. You can see the original Haskell and the resulting Coq in Figure 7.5. In GHC, the same `Var` type is used to represent type and term variables; as we are only concerned with term-level behavior and scoping, we use the `skip constructor` edit to eliminate the `TyVar` and `TcTyVar` constructors. Furthermore, the `IdInfo` type contains a great deal of metadata about the variable, not all of which we need and some of which we cannot translate. Some of the information in the `id_info` field includes *unfoldings* – the value of the variable, exposed for inlining optimizations. As we discussed above in Section 7.5, this means that the `id_info` field makes reference to other `Var`s in those unfoldings, which themselves may have `id_info`s that reference `Var`s – and in fact, this structure can be cyclic! This sort of infinite data is perfectly common in Haskell, but forbidden in Coq, so there is even more simplification that goes on behind the scenes.

We also see the presence of a "unique". This is an (unboxed) integer used to make comparison efficient – if two `Var`s have different uniques, they are different.

```
data Var
  = TyVar {     -- Type and kind variables
        varName    :: !Name,
        realUnique :: {-# UNPACK #-} !Int,
        varType    :: Kind }
  | TcTyVar {   -- Used only during type inference
        varName        :: !Name,
        realUnique     :: {-# UNPACK #-} !Int,
        varType        :: Kind,
        tc_tv_details  :: TcTyVarDetails  }
  | Id {
        varName    :: !Name,
        realUnique :: {-# UNPACK #-} !Int,
        varType    :: Type,
        idScope    :: IdScope,
        id_details :: IdDetails,
        id_info    :: IdInfo }
```

```
Inductive ...
  (* part of a mutually inductive type *)
with Var : Type
  := | Mk_Id (varName    : Name.Name)
             (realUnique : BinNums.N)
             (varType    : Type_)
             (idScope    : IdScope)
             (id_details : IdDetails)
             (id_info    : IdInfo)  : Var
```

FIGURE 7.5. The representation of variables in GHC, and its conversion into Coq.

Unfortunately, the name is a lie: it is an explicitly-documented requirement that two different variables can have the same unique. Additionally, the name `realUnique` is used because the `Name` field also contains the same unique – it is stored in both places for efficiency. This sort of duplication also shows up when determining if this variable is local or global; that information is stored both in the `idScope` and in the unique itself. Thus, to ensure that our variables are well-formed, we need more than just "every variable is bound" – first, the variables must satisfy these rules. We represent this with the Coq proposition `GoodVar : Var -> Prop`:

```
Definition GoodVar (v : Var) : Prop :=
  isLocalVar v = isLocalScope v /\
  varUnique v = nameUnique (varName v).
```

While at the end of the compilation pipeline, this may change (for instance, the last Core-to-Core pass makes all variable scopes global), this invariant holds of all variables that we look at.

Once we know this, we can talk about what it means for a variable to be well-scoped. Well-scoped variables are those `GoodVar`s that lie within the currently in-scope set of variables. However, there are again wrinkles around uniques. In particular, GHC has a type `VarSet` of sets of variables, and this type is indexed by the *unique*. Again, however, this is not truly unique, so we need to check that the variable we get out is the *right* variable with the same unique. One might think an equality check would work, but alas no: the `id_info` field of the record may be updated during an optimization pass to store information, but this does not change which variable is present. We thus define a relation `almostEqual : Var -> Var -> Prop` which asserts that two `Var`s are equal in every other field, and can then say what it means to be a `WellScopedVar`:

```
Definition WellScopedVar (v : Var) (in_scope : VarSet) : Prop :=
  if isLocalVar v then
   match lookupVarSet in_scope v with
    | None => False
    | Some v' => almostEqual v v' /\ GoodVar v
   end
```

7.8.2. Join points. Join points were a recent addition to GHC (Maurer, Downen, Ariola, and Peyton Jones, 2017), in part by Joachim Breitner, one of my collaborators. They can be viewed as a form of enhanced tail calls: a *join point* is a special kind of function that can only be called in tail position, and must be fully applied. This way, operationally, a call to a join point is simply a jump instruction, rather than a full function call.

As we saw in Figure 7.2, there is no distinct syntactic category for join points – instead, join points are simply distinguished variables. As such, they need to be used appropriately; GHC requires the following invariants hold of join points, as documented in a comment in the module `CoreSyn`:

> 1. All occurrences must be tail calls. Each of these tail calls must pass the same number of arguments, counting both types and values; we call this the "join arity" (to distinguish from regular arity, which only counts values).
> 2. For join arity n, the right-hand side must begin with at least n lambdas. No ticks, no casts, just lambdas! C.f. `CoreUtils.joinRhsArity`.
> 2a. Moreover, this same constraint applies to any unfolding of the binder. Reason: if we want to push a continuation into the RHS we must push it into the unfolding as well.
> 3. If the binding is recursive, then all other bindings in the recursive group must also be join points.
> 4. The binding's type must not be polymorphic in its return type (as defined in Note `[The polymorphism rule of join points]`).
>
> —GHC, the Note `[Invariants on join points]` in the module `CoreSyn`, lines 592–607

Invariant 1 tells us how join points must be used, and specifies that this is the *only* way they can be used – they cannot even be bound to other variables, much less passed to higher-order functions. Invariant 2 tells us that the join point must be a literal function – no point-free applications, no profiling information (`Tick`), no casts. This builds on the concept of "join arity" from invariant 1: a join point `j_f` may have type `A -> B -> C -> D`, but join arity `2`. In this case, all calls must be of the form `j_f a b`, and the definition can be `j_f = \a -> \b -> g a b` – note the two lambdas, followed by `g a b :: C -> D`. Invariant 3 tells us that join points may not be mutually recursive with normal functions – we must keep the two worlds separate for efficiency.

We cannot, however, validate invariants 2a or 4. Invariant 2a discusses "unfoldings" – metadata about identifiers used for optimization, so that values can be replaced by their definitions. We elide this sort of metadata to break some infinite self-reference in the definition of `Core`, so we cannot prove anything about it. Invariant 4 deals with type information, which we similarly elide.

7.8.3. The theorems. Our final main proofs were the following two theorems:

```
Lemma WellScoped_substExpr : forall e s vs subst,
    WellScoped_Subst subst vs ->
    WellScoped e vs ->
    WellScoped (substExpr s subst e)
               (getSubstInScopeVars subst).

Theorem exitifyProgram_WellScoped_JPV:
  forall pgm,
  WellScopedProgram pgm ->
  isJoinPointsValidProgram pgm ->
  WellScopedProgram (exitifyProgram pgm) /\
  isJoinPointsValidProgram (exitifyProgram pgm).
```

The former, `wellscoped_substExpr`, states that substitution on terms, `substExpr`, preserves well-scopedness of variables. The latter, `exitifyProgram_WellScoped_JPV`, states that the *exitify* optimization pass preserves the conjunction of well-scopedness and the join point invariants.

The substitution result is broadly clear, although the particular formulation of the theorem is tricky. In particular, substitutions in GHC maintain a set of variables that *will* be in scope – all the variables that will be present once the substitution has been applied. (We defined the function `getSubstInScopeVars` to perform the necessary composition of projections to access this set.) In other words, this set is for the *range* of the substitution, not the domain.

Exitification is a more full-fledged optimization pass, designed, essentially, to find opportunities for inlining optimizations by identifying the base cases of recursive functions (the *exit path*). Inlining is tricky because it can be both an optimization and a pessimization: if we inline the definition of a variable into a function, and the function is called only once, then this is a win. If the function is called many times,

this is only a win if computing the value of the variable is fast enough, which the compiler cannot determine. Thus, in general, GHC usually avoids doing this inlining.

However, join points are always tail calls, and so, in the simple case, inlining into one is always safe – once it's called, there are no more chances to call it! Except for in one case: if the join point is recursive. Then what? The answer is that in that case, the base case of the recursion is *still* only called once – this is what it means to be the base case. Once this case is found, it can be abstracted into a single join point, and then inlining into that join point can be automatically done (by a later optimization pass).

7.9. Gradations of being live

As we've applied these new edits and techniques to the large problem of verifying GHC, we've had cause to reconsider our approach. The theme of the past three chapters (Chapter 5, Chapter 6, and this current chapter) has been broadening the scope of this project in two ways: first, broadening the scope of what `hs-to-coq` can verify; but second and more interestingly, broadening our perspective on what a good translation looks like. When we started working with `hs-to-coq`, our goal was to produce translated code that was as close to an exact copy of the Haskell code as possible – part of DeepSpec's notion of being *live* (Appel et al., 2017). But as the size of the goals we set ourselves broadened, we realized that this was too restrictive. The key insight, which was due to collaborative effort by the whole `hs-to-coq` team, was that we were still learning something valuable even while adjusting the semantics of the translated code. Back in Chapter 5, we were focused on being "exactly live" in this way. In Chapter 6, when working on `containers`, we loosened our requirements to require identical semantics at the top level, but were more flexible with function internals. What we have seen in this chapter, with verifying parts of GHC, is the natural evolution: relaxing the semantics to focus on the details we care about.

One way to see why we needed to take this step is to look at the scale of GHC. GHC 8.4.3 contains 182 174 (nonblank noncomment) lines of Haskell code[29] and 464 modules, a full order of magnitude larger than our previous largest verification target, the `containers` library. What's more, it was far more *entangled*, with a complex dependency graph and mutual recursion between modules; we saw the mutually recursive portion of this graph in Figure 7.3, but this leaves out everything else. We translate 99 modules comprising 29 703 lines of Haskell, or roughly 16 % of GHC; this generates 22 715 lines of Coq code. (Here and going forward, all line counts refer only to nonblank, noncomment lines of code.)

On the other hand, `containers`, which was no small potatoes, was nevertheless much more manageable, both numerically and conceptually. Numerically speaking, the 7 modules we translated clocked in at 6 596 lines of Haskell, of which we skipped very little; this is an order of magnitude smaller than our translation of GHC. Conceptually speaking, the dependency graph of `containers` is significantly simpler than that of GHC; instead of a single compiler, the library defines seven different data structures, all of which are relatively orthogonal to each other. This means that, while the full

[29] This and all other line counts were computed with Al Danial's `cloc` tool, available at https://github.com/AlDanial/cloc.

scope of `containers` is an order of magnitude larger than what we translated – it has an additional 41 modules and 10 485 lines of Haskell, for a total of 48 modules and 17 081 lines of Haskell[30] – we did not need to think about the modules we did not translate beyond skipping four utility modules. When translating GHC, on the other hand, figuring out which modules to skip (and which modules to axiomatize) was a challenge in and of itself. This added up to make GHC a much more complicated code base to deal with than `containers` was: not only did we have an order of magnitude more code to look at than when translating GHC than when translating `containers`, and not only did we then translate an order of magnitude more code, figuring out which modules contained the code we needed to translate in the first place was also a significantly more difficult process.

As this comparison helps demonstrate, without adopting this more flexible approach, we would never have been able to verify the portions of GHC that we did. GHC was simply too large, too partial, too *Haskell* to otherwise be compatible with Coq verification. And our experience conducting this verification leaves us confident that the results are meaningful. In almost all verification settings, after all, we appreciate results that verify approximations of the running code: verifying models, verifying trusted kernels, using SMT solvers for portions of code, and similar such things. With Coq, we do not expect this gap, and tend to expect total coverage and correctness. But there is nothing intrinsic about Coq that requires this, *assuming* that we can get our partial model in the first place. And it is exactly this which `hs-to-coq` allows: producing a model in Coq of Haskell code, and allowing the user to configure the precision of that model by using edits.

While this approach is valuable, it – like everything else – has tradeoffs. The major downsides to this approach is that we lose the comprehensive guarantees that being fully live give you, and weaken the "deepness" of our specification. But the benefits are more than worth it. We can scale our verification effort based on what our target is. We can verify larger codebases by slicing out the portion we care about. We can evolve our verification over time, starting by ignoring as much code as we need to and then adding more verification in over time. We can verify either kernels that need to be trusted, or we can assume that a trusted kernel is correct and verify what lies on top of it. In short, we get *flexibility*, both between projects and within a project, and we still get enough guarantees for the result to be worth it.

To sum up, live verification – by which we here mean verifying the exact code that's running – is a valuable *goal*, but fundamentally impossible for some codebases. But writing down the differences and aiming for "almost entirely live" remains valuable! And freeing ourselves from that exactness, while still retaining "as live as we can" as something to think about, opens up Coq for use as a verification tool in particular ways that had never available before.

[30] Because we used testing code as part of our translation, these numbers include the tests as well as the regular modules. Breaking down our numbers along these lines: of the 7 modules we translated, 5 were regular and 2 were from the tests, for 6 214 and 382 lines of Haskell, respectively; of the remaining 41 modules, 30 were regular and 11 were from the tests, for 6 875 and 3 610 lines of Haskell, respectively; and in total, of all 48 modules, 35 were regular and 13 were from the tests, for 13 089 and 3 992 lines of Haskell, respectively.

CHAPTER 8

A Comprehensive Exposition of the Edit Language

In the past three chapters, we have seen how `hs-to-coq` can be used to verify Haskell code; along the way, we have seen the wide array of features of `hs-to-coq`'s edit language, and how that language was built up over time. It is now time, in this chapter, to provide a detailed reference for the whole edit language. After covering the detailed syntax of edits (Section 8.1), I present an explanation of each of the 34 distinct edits that `hs-to-coq` supports.[31] Each of these examples contains the following pieces:

(1) The syntax of the edit.
(2) A written explanation of its behavior.
(3) A concrete example of how the edit behaves when applied to a piece of Haskell code, showing the Haskell and edit file inputs and the translated Coq output.

We present these edits according to the same categories that were outlined in Chapter 4: eight categories of edits divided by their purpose, as well as the "meta-edit" `in`. As a refresher, these are the categories of edit purposes that we saw in Section 4.1:

- Skipping Haskell code (Section 8.2);
- Axiomatizing Haskell code (Section 8.3);
- Adding Coq code (Section 8.4);
- Changing the structure of the Haskell code (Section 8.5);
- Rewriting expressions (Section 8.6);
- Providing extra information (Section 8.7);
- Proving termination (Section 8.8); and
- Meta-edits (Section 8.9).

8.1. The syntax of edits

Each individual edit begins with a sequence of keywords that say which edit it is; for example, above, we saw **`skip module`**, **`skip class`**, **`rename`**, and **`add`**. These demonstrate the following points:

- Edit names can be multiple words (e.g., **`skip module`**);
- Edit names can share words, even the first word, and be distinct (e.g., **`skip module`** vs. **`skip class`**); and
- Edit names can be followed by further non-name keywords, (e.g., **`rename type`** vs. **`rename value`**).

[31] Some of these explanations are based on documentation I wrote for `hs-to-coq`, which is available at https://hs-to-coq.readthedocs.io/en/latest/edits.html; the contents of that site, however, may have changed since the publication of this dissertation.

We also saw that edits can have more syntax after the edit name; for example, `rename` is followed by a Haskell namespace (`type` or `value`), a Haskell name, an equals sign (`=`), and a Coq name.

Edits are in general composed of ten kinds of lexemes: keywords, identifiers, operators, numbers, strings, opening delimiters, closing delimiters, Ltac tactics, Coq proof terminators, and newlines.

Keywords are identifiers or operators that have a special meaning in edit files. They come in three classes: portions of edit names, such as `skip` or `module`; edit keywords, which are used within edits, such as `=` or `value`; and a subset of Coq keywords and syntax, such as `Inductive` or `:=`. There is some overlap between these categories; for example, `:` is both an edit keyword and a Coq keyword.

Identifiers, also called *Words*, are programming-language style identifiers – alphanumeric sequences, such as `foldr` or `liftA2`. More precisely, as in Haskell or Coq, these names can contain all alphanumeric characters, underscores, and apostrophes (primes), although they must start with a letter or an underscore; similar to the GHC Haskell extension `MagicHash`, they can also contain `#`. Furthermore, these identifiers can be *module-qualified*, such as `Data.Map.Strict.fromList`. A qualified name in this context is as in Haskell: a sequence of unqualified names separated with periods, all but the last of which must start with an uppercase letter.

Operators are programming-language style infix operators – sequences of general punctuation characters, such as `+` or `=<<`. We also include some less standard strings in this category: the Haskell tuple operators in parentheses, such as `(,)` and `(,,)`, and the special tokens `()` and `[]` (containing optional space). Just as identifiers can be fully qualified, so can operators, such as `Data.Set.\\`.

Numbers are natural numbers represented as strings of decimal digits, such as `42` or `000`.

Strings are arbitrary sequences of non-quote characters enclosed in double quotes, such as `"Hello, world!"`. There are no escape sequences; `"no \n"` is a five-character string that begins and ends with an `n`.

Opening delimiters are characters such as `(` and `{`. They are paired with closing delimiters

Closing delimiters are characters such as `}` and `)`. They are paired with opening delimiters.

Ltac tactics are arbitrary sequences of text – while one edit attempts to parse a very limited subset of Ltac, we do not attempt to parse its full generality as used in proof bodies. These sequences are stopped by one of the Coq proof terminators.

Coq proof terminators are the three ways you can end proofs in Coq: `Qed`, `Defined`, or `Admitted`. (The period comes from the surrounding context.) These strings cannot occur inside Ltac tactics, but are not treated specially except for their use in terminating Ltac.

Newlines are just that, or more properly any sequence of vertical whitespace. They generally terminate edits, although some edits take arbitrary Coq as a parameter, and the Coq code is permitted to extend through line breaks.

Thus, for example, I represent the syntax of the `skip module` edit seen above as `skip module` *`Identifier`*, where the leading uppercase letter and italics to indicate a metavariable.

As expected for a programming language, we of course extend the grammar to include more complex structure, but we do not present the whole thing here. We do, however, present some particularly common ones:

- *Qualids* are any possibly-qualified name; that is, either an identifier or an operator.
- *Patterns* are a subset of the Coq pattern language, containing constructor application, variables, underscores, numbers, and `as`-patterns. `AtomicPatterns` are those patterns which are either a single identifier number or underscore, or parenthesized.
- *CoqDefinitions* are Coq definitions. They are given by one of the following Coq Vernacular commands: `Inductive`, `CoInductive`, `Definition`, `Let`, `Fixpoint`, `CoFixpoint`, `Instance`, `Axiom`, `Theorem`, `Lemma`, `Remark`, `Fact`, `Corollary`, `Proposition`, or `Example` (the latter seven coming with `Proof`s.) These definitions, unlike other portions of edits, may contain newlines; they are terminated with a period. Our grammar of Coq is a very limited subset of the full grammar, does not understand precedence, and does not include all of the Coq's syntactic sugar, so some rewriting of terms may be necessary.

Now that we have seen the syntax of edits, we can turn to documenting each individual edit in turn.

8.2. Skipping Haskell code

8.2.1. skip.

Edit: `skip` *`Qualid`*

Effect: This edit suppresses the translation of a single value or type. The value or type being skipped is given by its translated Coq name, so operator names must be expanded and renamings have already been applied.

This edit cannot skip subcomponents of larger definitions, and nor can it skip type classes. It *can*, however, skip type class instances, which are simply value definitions in Coq. Instance names are deterministic, so to find the name of an instance to skip, `hs-to-coq` can be run once without the `skip` edit to find the name.

Examples.
Input:

```
# Edits
skip Data.Set.some_function

-- Haskell
module Data.Set where

some_function  = ...
```

```
other_function = ...
```

Output:

```
(* Coq *)
Definition other_function := ...
```

Input:

```
# Edits
skip Data.Set.Eq___Set_

-- Haskell
module Data.Set where

data Set a = ...
           deriving (Eq)
```

Output:

```
(* Coq *)
Inductive Set_ a := ...

(* No `Instance` for `Eq_` *)
```

8.2.2. skip constructor.

Edit: `skip constructor` *`Qualid`*

Effect: This edit takes the name of a single constructor of a data type, such as `GHC.Base.Just`, and completely omits it from the translated code: it is not present in the translated data type, and any branches of a pattern match that match on it are eliminated. Just as with `skip`, it does *not* remove uses of the constructor as a function, which must be handled separately. Because of this, however, a successful `skip constructor` edit does indicate that the translated code does not depend on the skipped constructor.

Examples.

Input:

```
# Edits
skip constructor Mod.Impure

-- Haskell
module Mod where

data Reference a = Pure a
                 | Impure (IO a)

isPure :: Reference a -> Bool
```

```
isPure (Pure   _) = True
isPure (Impure _) = False

impurelyPure :: a -> Reference a
impurelyPure x = Impure (pure x)
```

Output:

```
(* Coq *)
Inductive Reference a : Type := | Pure : a -> Reference a.

Arguments Pure {_} _.

(* Converted value declarations: *)

Definition isPure {a} : Reference a -> GHC.Types.Bool :=
  fun '(Pure _) => GHC.Types.True.

Definition impurelyPure {a} : a -> Reference a :=
  fun x => Impure (GHC.Base.pure x).
```

8.2.3. skip class.

Edit: `skip class` *Qualid*

Effect: This edit takes the name of a type class, such as `GHC.Show.Show`[32], and omits both the definition and all future instances of it from the translated code.

Examples.

Input:

```
# Edits
skip class GHC.Show.Show

-- Haskell
module GHC.Show where

-- ...

class Show a where
  -- ...

-- ...

module Mod where
```

[32]This is a translated name.

```
data T = C
instance Show T where show C = "C"
```

Output:

```
(* Coq *)

(* GHC/Show.v *)

(* ... *)

(* Mod.v *)

Inductive T : Type := | C : T.
```

8.2.4. skip method.

Edit: `skip method` *Qualid Word*

Effect: This edit takes the name of a type class, such as `GHC.Base.Alternative`, and the (unqualified) name of one of its methods, such as `some`, and both omits that method from the translated class definition and omits all future implementations of it. Any default implementation of that method is also omitted, both from the definition and from instances.

Examples.

Input:

```
# Edits
skip method GHC.Base.Alternative some
skip method GHC.Base.Alternative many

-- Haskell
module GHC.Base where

-- ...

class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

  some :: f a -> f [a]
  some = ...

  many :: f a -> f [a]
  many = ...
```

```
-- ...

module Mod where

data Opt a = No | Yes a

instance Functor Opt where
  fmap _ No      = No
  fmap f (Yes x) = Yes $ f x

instance Applicative Opt where
  pure = Yes

  No    <*> _     = No
  _     <*> No    = No
  Yes f <*> Yes x = Yes $ f x

instance Alt Opt where
  empty = No

  No        <|> o = o
  y@(Yes _) <|> _ = y

  some = ...
```

Output:

```
(* Coq *)

(* GHC/Base.v *)

(* ... *)

(* Mod.v *)

Inductive Opt a : Type := | No : Opt a |  Yes : a -> Opt a.

(* ... *)

Local Definition Alternative__Opt_empty : forall {a}, Opt a :=
  fun {a} => No.
```

```
Local Definition Alternative__Opt_op_zlzbzg__
    : forall {a}, Opt a -> Opt a -> Opt a :=
  fun {a} =>
    fun arg_0__ arg_1__ =>
      match arg_0__, arg_1__ with
      | No, o => o
      | (Yes _ as y), _ => y
      end.

Program Instance Alternative__Opt : Alternative Opt :=
  fun _ k__ =>
    k__ {| empty__ :=
             fun {a} => Alternative__Opt_empty ;
           op_zlzbzg____ :=
             fun {a} => Alternative__Opt_op_zlzbzg__ |}.
```

8.2.5. skip equation. (Originally from the documentation.[33])

Edit: `skip equation` *Qualid AtomicPattern+*

Effect: `skip equation` `qualified_function` `pattern...` skips the equation of the function definition whose arguments are the specified patterns. Guards are not considered, only the patterns themselves.

For example, consider the following (silly) function definition:

```
redundant :: Maybe Bool -> Maybe Bool -> Bool
redundant (Just True)  _        = False
redundant (Just False) _        = True
redundant _            _        = True
redundant _            (Just b) = b
```

The last case is redundant, so Coq will reject this definition. However, we can add the following edit:

```
skip equation ModuleName.redundant _ (Some b)
```

And the last case will be deleted on the Coq side:

```
Definition redundant : option bool -> option bool -> bool :=
  fun arg_0__ arg_1__ =>
    match arg_0__, arg_1__ with
    | Some true, _ => false
    | Some false, _ => true
    | _, _ => true
    end.
```

[33] Available at https://hs-to-coq.readthedocs.io/en/latest/edits.html#skip-equation-skip-one-equation-of-a-function-definition

Note that you have to use the translated name (`Some` vs. `Just`), and most constructor names will be fully qualified.

Why would you want this? This edit is most useful in tandem with the `skip constructor` edit (Section 8.2.2). Suppose we have a function where the final catch-all case can only match skipped constructors, such as

```
data T = TranslateMe
       | SkipMe

function :: T -> Bool
function TranslateMe = True
function _           = False
```

Then, on skipping `SkipMe`, this function's `_` case will be redundant, and Coq would reject it. We can fix this with

```
skip equation ModuleName.function _
```

to translate just the `TranslateMe` case.

See also `skip case pattern` for the equivalent edit for `case` and lambda-case expressions.

8.2.6. skip case pattern.

(Originally from the documentation.[34])

Edit: `skip case pattern` *Pattern*

Effect: `skip case pattern pat` skips any alternative of a `case` expression (or a lambda-case expression) which matches against the given `pat`tern. Guards are not considered, only the pattern itself.

For example, consider the following (silly) function definition:

```
redundant :: Bool -> Bool
redundant b = not (case b of
                     True  -> False
                     False -> True
                     _     -> True)
```

The last case is redundant, so Coq will reject this definition. However, we can add the following edit:

```
in ModuleName.redundant skip case pattern _
```

And the last case will be deleted on the Coq side (reformatted):

```
Definition redundant : bool -> bool :=
  fun b => negb (match b with
                 | true => false
                 | false => true
                 end).
```

[34] Available at https://hs-to-coq.readthedocs.io/en/latest/edits.html#skip-case-pattern-skip-one-alternative-of-a-case-expression

You can use an arbitrary pattern, not simply `_`; constructor names must be fully qualified and the names used must be those that appear *after* renaming.

Why would you want this? This edit is most useful in tandem with the **skip constructor** edit (Section 8.2.2); see the discussion in **skip equation** for a worked example (with a named function).

This edit is unusual in that you *very likely* want to use it with the **in** meta-edit (Section 8.9.1) to scope its effects to within a specific definition. However, this isn't mandatory; if, for some reason, you want to skip every `_` in every **case**, then **skip case pattern** `_` will do what you want.

See also **skip equation** for the equivalent edit for named functions.

8.2.7. skip module.

Edit: **skip module** *Word*

Effect: This edit takes the name of a Coq module, and refrains from generating any **Require** statements for it. If in recursive translation mode, `hs-to-coq` will also refrain from translating this module if it otherwise would.

In non-recursive mode, this edit will not generally be useful: `hs-to-coq` automatically detects all the modules that its names come from, and **Require**s them automatically. However, during development, it may be useful to allow translation of a prefix of a file, prior to any missing modules, and this edit enables that developmental workflow.

Examples.

Input:

```
# Edits
skip module GHC.Show

-- Haskell
module Mod where

-- Show is from GHC.Show
showTwice :: Show a => a -> String
showTwice x = show x ++ show x
```

Output:

```
(* Coq *)

Require GHC.Base.
Import GHC.Base.Notations.

Definition showTwice {a} `{GHC.Show.Show a}
    : a -> GHC.Base.String :=
  fun x => GHC.Show.show x GHC.Base.++ GHC.Show.show x.
```

8.3. Axiomatizing Haskell code

8.3.1. axiomatize module.

Edit: `axiomatize module` *`Word`*

Effect: This edit takes the name of a Coq module and changes how it is translated. Instead of translating the types and values as normal, the types and type class definitions are translated and the values are replaced with axioms having the appropriate type. This is useful for stubbing out a module during development, or for taking an untranslated (or even untranslatable) module and treating it as part of the trusted code base. To replace types with axioms as well, use the **`axiomatize definition`** edit (Section 8.3.3).

Examples.
Input:

```
# Edits
axiomatize module Mod

-- Haskell
module Mod where

data Pair a = MkPair a a

diag :: Pair (Pair a) -> Pair a
diag (MkPair (MkPair x _) (MkPair _ y)) = MkPair x y

fix :: (a -> a) -> a
fix f = f (fix f)
```

Output:

```
(* Coq *)

(* Converted type declarations: *)

Inductive Pair a : Type := | MkPair : a -> a -> MkPair a.

Arguments MkPair {_} _ _.

(* Converted value declarations: *)

Axiom fix_ : forall {a}, (a -> a) -> a.

Axiom diag : forall {a}, Pair (Pair a) -> Pair a.
```

8.3.2. axiomatize original module name. (Originally from the documentation.[35])

Edit: `axiomatize original module name` *`Word`*

Effect: You probably do not need to use this edit; it's only important when using `rename module` to merge multiple modules into one. If you are doing this, however, and wish you could use `axiomatize module` on *some* of the input modules but not others, then `axiomatize original module name` is the edit for you.

The behavior of `axiomatize original module name` is the same as the behavior of `axiomatize module`, except for how it picks which module to axiomatize. While every other edit operates in terms of the Coq name after any renamings from the edits have been applied, `axiomatize original module name` checks the *original*, pre-`rename module`, form of the module name. Most of the time, this would be confusing, and `axiomatize module` would be preferable.

However, if you have used `rename module` to merge two (or more) modules into one, but you only want one of them (or some other strict subset) to be axiomatized, then `axiomatize original module name` is the only way to get this behavior.

Examples.
Input:

```
# Edits
axiomatize original module name Part1
rename module Part1 Whole
rename module Part2 Whole

-- Haskell
module Part1 where

data Type1 = Con1

axiom :: Type1
axiom = Con1

-- Haskell
module Part2 where

data Type2 = Con2

definition :: Type2
definition = Con2
```

[35] Available at https://hs-to-coq.readthedocs.io/en/latest/edits.html#axiomatize-original-module-name-replace-all-definitions-in-a-module-with-axioms-using-the-pre-renaming-module-name

Output:

```
(* Coq *)

(* Whole.v *)

(* Converted type declarations: *)

Inductive Type2 : Type := | Con2 : Type2.

Inductive Type1 : Type := | Con1 : Type1.

Instance Default__Type2 : GHC.Err.Default Type2 :=
  GHC.Err.Build_Default _ Con2.

Instance Default__Type1 : GHC.Err.Default Type1 :=
  GHC.Err.Build_Default _ Con1.

(* Converted value declarations: *)

Axiom axiom : Type1.

Definition definition : Type2 :=
  Con2.
```

8.3.3. axiomatize definition.

Edit: `axiomatize definition` *Qualid*

Effect: (Originally from the documentation.[36]) Replaces a single definition with an axiom. This takes the name of a value-level definition and, when translating it, translates only the type and generates an axiom with that type. See also `axiomatize module` (Section 8.3.1), and also `redefine Axiom` (Section 8.4.3) for type-level axiomatization.

Examples.
Input:

```
# Edits
axiomatize definition Mod.fix_

-- Haskell
module Mod where
```

[36] Available at https://hs-to-coq.readthedocs.io/en/latest/edits.html#axiomatize-definition-replace-a-value-definition-with-an-axiom

```
data Pair a = MkPair a a

diag :: Pair (Pair a) -> Pair a
diag (MkPair (MkPair x _) (MkPair _ y)) = MkPair x y

fix :: (a -> a) -> a
fix f = f (fix f)
```

Output:

```
(* Coq *)

(* Converted type declarations: *)

Inductive Pair a : Type := | MkPair : a -> a -> MkPair a.

Arguments MkPair {_} _ _.

(* Converted value declarations: *)

Axiom fix_ : forall {a}, (a -> a) -> a.

Definition diag {a} : Pair (Pair a) -> Pair a :=
  fun '(MkPair (MkPair x _) (MkPair _ y)) => MkPair x y.
```

8.3.4. unaxiomatize definition.

Edit: `unaxiomatize definition` *Qualid*

Effect: (Originally from the documentation.[37]) Translates a single definition, `axiomatize module` (Section 8.3.1) notwithstanding.

If the module containing the given value-level definition is being axiomatized, then this definition will be translated in the usual way.

If a definition is both `unaxiomatize`d and `skip`ped, then it will simply be skipped.

Examples.

Input:

```
# Edits
axiomatize module Mod
unaxiomatize definition Mod.diag
```

[37] Available at https://hs-to-coq.readthedocs.io/en/latest/edits.html#unaxiomatize-definition-override-whole-module-axiomatization-on-a-case-by-case-basis

```
-- Haskell
module Mod where

data Pair a = MkPair a a

diag :: Pair (Pair a) -> Pair a
diag (MkPair (MkPair x _) (MkPair _ y)) = MkPair x y

fix :: (a -> a) -> a
fix f = f (fix f)
```

Output:

```
(* Coq *)

(* Converted type declarations: *)

Inductive Pair a : Type := | MkPair : a -> a -> MkPair a.

Arguments MkPair {_} _ _.

(* Converted value declarations: *)

Axiom fix_ : forall {a}, (a -> a) -> a.

Definition diag {a} : Pair (Pair a) -> Pair a :=
  fun '(MkPair (MkPair x _) (MkPair _ y)) => MkPair x y.
```

8.4. Adding Coq code

8.4.1. add.

Edit: `add` *Word CoqDefinition*

Effect: Adds a Coq definition to the value section translation of a module. The module name is given as the first parameter to `add`, and the definition follows. The Coq parser is limited, but supports newlines and binary operators; the definition must end with a period. All names used in the definition must be fully qualified, including the name being defined; `hs-to-coq` will strip that module qualifier and make the term syntactically valid.

`Theorem`-like environments (`Theorem`, `Lemma`, `Remark`, `Fact`, `Corollary`, `Proposition`, and `Example`) are permitted, although Ltac is not parsed. Instead, each of them must be followed immediately by `Proof.`, some unparsed text (Ltac), and then any of `Qed`, `Defined`, or `Admitted`, followed by a period.

The inserted definition is placed into the module in dependency order.

This is one of three edits that can span multiple lines, as line breaks are permitted anywhere inside the Coq definition (but not before it); the other such edits, which also contain Coq definitions, are **`add type`** (Section 8.4.2) and **`redefine`** (Section 8.4.3).

Examples.

Input:

```
# Edits
add Mod Definition Mod.diag {a}
    : Mod.Pair (Mod.Pair a) -> Mod.Pair a :=
  fun p => match p with
    | (Mod.MkPair (Mod.MkPair x _) (Mod.MkPair _ y)) =>
      Mod.MkPair x y
  end.

add Mod Theorem Mod.double_00 : Mod.Pair nat.
Proof.
  exact (MkPair 0 0).
Qed.

-- Haskell
module Mod where

data Pair a = MkPair a a
```

Output:

```
(* Coq *)

(* Converted type declarations: *)

Inductive Pair a : Type := | MkPair : a -> a -> Pair a.

Arguments MkPair {_} _ _.

(* Converted value declarations: *)

Theorem double_00 : Pair nat.
Proof.
  exact (MkPair 0 0).
Qed.

Definition diag {a} : Pair (Pair a) -> Pair a :=
  fun '(MkPair (MkPair x _) (MkPair _ y)) => MkPair x y.
```

8.4.2. add type.

Edit: `add type` *Word CoqDefinition*

Effect: Adds a Coq definition to the types-and-classes section of the translation of a module. This works just like **add** (Section 8.4.1), but places the definition amongst the types. The reason that `hs-to-coq` separates the types and classes is that, thanks to type inference, we cannot detect which types are used in a Haskell term from strictly syntactic examination. Consequently, we do separate dependency analysis among (1) types and classes and (2) values. The choice of **add** vs. **add type** is about which universe you would like your inserted term to live in.

This is one of three edits that can span multiple lines, as line breaks are permitted anywhere inside the Coq definition (but not before it); the other such edits, which also contain Coq definitions, are **add** (Section 8.4.1) and **redefine** (Section 8.4.3).

Examples.
Input:

```
# Edits
add type Mod Inductive Mod.Triple a :=
  | Mod.MkTriple : a -> (a -> (a -> Mod.Triple a)).

add Mod Definition Mod.diag3 {a} (t : Mod.Triple (Mod.Triple a))
                     : Mod.Triple a :=
  match t with
  | Mod.MkTriple (Mod.MkTriple x _ _)
                 (Mod.MkTriple _ y _)
                 (Mod.MkTriple _ _ z) =>
    Mod.MkTriple x y z
  end.


-- Haskell
module Mod where

data Pair a = MkPair a a

diag :: Pair (Pair a) -> Pair a
diag (MkPair (MkPair x _) (MkPair _ y)) = MkPair x y


(* Coq *)
(* midamble.v *)

Arguments MkTriple {_} _ _ _.
```

Output:

```
(* Converted type declarations: *)

Inductive Pair a : Type := | MkPair : a -> a -> Pair a.

Arguments MkPair {_} _ _.

Inductive Triple a : Type := | MkTriple : a -> a -> a -> Triple a.

(* Converted value declarations: *)

Definition diag {a} : Pair (Pair a) -> Pair a :=
  fun '(MkPair (MkPair x _) (MkPair _ y)) => MkPair x y.

Definition diag3 {a} (t : Triple (Triple a)) : Triple a :=
  let 'MkTriple (MkTriple x _ _) (MkTriple _ y _) (MkTriple _ _ z)
        := t in
  MkTriple x y z.
```

8.4.3. redefine.

Edit: `redefine` *CoqDefinition*

Effect: Replaces one Coq definition with another. The identifier that is the name of the given definition, which must be fully-qualified and include a module name, is monitored for and, if it would be translated, the given redefinition is used instead.

This can also be viewed as a combination of the `skip` (Section 8.2.1) and `add`/`add type` (Section 8.4.1/Section 8.4.2) edits.

This edit can be used, among other things, to replace type definitions with axioms; for example, consider Haskell's `IORef` type, which represents real mutable variables. Coq doesn't support this feature, so `IORef` could be translated by redefining it into an axiom with `redefine Axiom Data.IORef.IORef : Type -> Type.`. However, for redefining term-level definitions, please prefer `axiomatize definition` (Section 8.3.3), which preserves the type of the term. (We cannot do this automatically for type-level terms because GHC's kind inference is both more aggressive and more commonly used.)

This is one of three edits that can span multiple lines, as line breaks are permitted anywhere inside the Coq definition (but not before it); the other such edits, which also contain Coq definitions, are `add` (Section 8.4.1) and `add type` (Section 8.4.2).

Examples.

Input:

```
# Edits
redefine Definition Mod.display {a} (_ : Pair a)
    : GHC.Base.String :=
  GHC.Base.hs_string__ "".
  # We don't deal with the `Show` class

-- Haskell
module Mod where

data Pair a = MkPair a a

display :: Show a => Pair a -> String
display (MkPair x y) = show x ++ " & " ++ show y

diag :: Pair (Pair a) -> Pair a
diag (MkPair (MkPair x _) (MkPair _ y)) = MkPair x y
```

Output:

```
(* Converted type declarations: *)

Inductive Pair a : Type := | MkPair : a -> a -> Pair a.

Arguments MkPair {_} _ _.

(* Converted value declarations: *)

Definition display {a} (_ : Pair a) : GHC.Base.String :=
  GHC.Base.hs_string__ "".

Definition diag {a} : Pair (Pair a) -> Pair a :=
  fun '(MkPair (MkPair x _) (MkPair _ y)) => MkPair x y.
```

8.4.4. import module.

Edit: `import module` *`Word`*

Effect: If the given module is required by the translated Coq source code, it will be imported directly, and references to the names it contains will be unqualified. This edit is only necessary for making the Coq output easier to read, but it can do an effective job of that.

When using this edit, it is important to remember that Coq does not behave like Haskell: it has no notion of import or export lists, and its notion of module re-exporting is completely different (in particular, re-exporting a name in Haskell introduces a new qualified name, whereas Coq's `Export` command does not). You

may have to pick a module name further up the hierarchy to import (for example, `import module Prelude` won't do anything, as all the definitions are in other modules re-exported by `Prelude`), and you will get all names that would otherwise be hidden.

Examples.
Input:

```
# Edits
# No `import module`

-- Haskell
module Mod where

import Control.Monad

flatten3 :: [[[a]]] -> [a]
flatten3 = join . join
```

Output:

```
(* Converted imports: *)

Require GHC.Base.
Import GHC.Base.Notations.

(* No type declarations to convert. *)

(* Converted value declarations: *)

Definition flatten3 {a} : list (list (list a)) -> list a :=
  GHC.Base.join GHC.Base.∘ GHC.Base.join.
```

Input:

```
# Edits
import module GHC.Base

-- Haskell
module Mod where

import Control.Monad

flatten3 :: [[[a]]] -> [a]
flatten3 = join . join
```

Output:

```
(* Converted imports: *)

Require Import GHC.Base.

(* No type declarations to convert. *)

(* Converted value declarations: *)

Definition flatten3 {a} : list (list (list a)) -> list a :=
  join ∘ join.
```

8.4.5. type synonym.

Edit: `type synonym Word :-> Word`

Effect: Specify the result type of the given type synonym. Sometimes, Haskell code specifies type synonyms where GHC's kind inference is more cooperative than Coq's. This edit allows you to tell `hs-to-coq` the result kind of a type synonym, which it may otherwise be unable to figure out. This does not allow the user to specify the kinds of the arguments.

Examples.
Input:

```
# Edits
type synonym Three :-> Type

-- Haskell
module Mod where

type Two   a = (a,a)
type Three a = (a,a,a)
```

Output:

```
(* Coq *)
Definition Two a :=
  (a * a)%type%type.

Definition Three a : Type :=
  (a * a * a)%type%type.
```

8.4.6. add scope.

Edit: `add scope Scope for ScopePlace Qualid`

Effect: Adds a Coq scope annotation to the given term. The scope place, which can be `constructor` or `value`, tells `hs-to-coq` the sort of term to annotate. The scope itself can be any word, including the usually-reserved word `type` as this is a common scope in Coq and only reserved for Haskell reasons.

Examples.

Input:

```
# Edits
add scope ctor_scope for constructor Mod.C
add scope val_scope  for value       Mod.f

-- Haskell
module Mod where

data T a b = C a b
           | D b a

f :: T a b -> T b a
f (C a b) = C b a
f (D b a) = D a b
```

Output:

```
(* Converted type declarations: *)

Inductive T a b : Type
  := | C : (a -> b -> T a b)%ctor_scope
  |  D : b -> a -> T a b.

Arguments C {_} {_} _ _.

Arguments D {_} {_} _ _.

(* Converted value declarations: *)

Definition f {a} {b} : T a b -> T b a :=
  (fun arg_0__ =>
     match arg_0__ with
     | C a b => C b a
     | D b a => D a b
     end)%val_scope.
```

8.5. Changing the structure of the Haskell code

8.5.1. collapse let.

Edit: `collapse let` *`Qualid`*

Effect: (Originally from the documentation.[38])If a converted definition is of the form

```
Definition outer := let inner := definition in inner.
```

then convert it to simply

```
Definition outer := definition.
```

Both `outer` and `inner` can have arguments; `inner` can have a type annotation, but it's ignored.

Additionally, if `definition` is a non-mutual fixpoint `fix f args := body`, the recursive calls to `f` in `body` are rewritten to direct calls to `outer`. This is particularly important for mutual recursion: if `inner` is mutually recursive with another top-level function, then if `outer` has no arguments, it would otherwise appear not to be a function and would thus cause conversion to fail, as Coq doesn't support recursion through non-functions.

Examples.
Input:

```
# Edits
collapse let Mod.map


-- Haskell
module Mod where

-- Two identical copies of `map`

map_plain :: (a -> b) -> [a] -> [b]
map_plain = go where
  go f []     = []
  go f (x:xs) = f x : go f xs

map_collapse :: (a -> b) -> [a] -> [b]
map_collapse = go where
  go f []     = []
  go f (x:xs) = f x : go f xs
```

[38]Available at https://hs-to-coq.readthedocs.io/en/latest/edits.html#collapse-let-if-a-definition-is-just-a-let-expression-inline-it

Output:

```
(* Coq *)

Definition map_plain {a} {b} : (a -> b) -> list a -> list b :=
  let fix go arg_0__ arg_1__
            := match arg_0__, arg_1__ with
               | f, nil => nil
               | f, cons x xs => cons (f x) (go f xs)
               end in
  go.

Definition map_collapse {a} {b} : (a -> b) -> list a -> list b :=
  fix map_collapse (arg_0__ : (a -> b)) (arg_1__ : list a) : list b
        := match arg_0__, arg_1__ with
           | f, nil => nil
           | f, cons x xs => cons (f x) (map_collapse f xs)
           end.
```

8.5.2. simple class.

Edit: `simple class` *Qualid*

Effect: Translates a Haskell type class directly into a Coq type class.

By default, `hs-to-coq`'s translation of type classes is (surprisingly) a CPSed encoding, as we saw back in Section 3.7. To reiterate the example from that section, the simple Haskell class and instance

```
class C a where
  m :: a
  f :: a -> Bool

instance C () where
  m   = ()
  f _ = True
```

is translated via a continuation-passing encoding into the following Coq code:

```
(* class C a where ... *)

Record C__Dict a := C__Dict_Build {
  f__ : a -> bool ;
  m__ : a }.

Definition C a :=
  forall r__, (C__Dict a -> r__) -> r__.
Existing Class C.
```

```
Definition f `{g__0__ : C a} : a -> bool :=
  g__0__ _ (f__ a).

Definition m `{g__0__ : C a} : a :=
  g__0__ _ (m__ a).

(* instance C Bool where ... *)

Local Definition C__unit_f : unit -> bool :=
  fun arg_0__ => true.

Local Definition C__unit_m : unit :=
  tt.

Program Instance C__unit : C unit :=
  fun _ k__ => k__ {| f__ := C__unit_f ; m__ := C__unit_m |}.
```

The simple record form of the type class is translated as `C__Dict`, and its methods have trailing double underscores; The type class `C` itself is the universally-quantified CPSed wrapper around `C__Dict`.

In Section 3.7, I mentioned that this encoding allows Coq to evaluate terms more easily, but gets in the way of encoding more complex Haskell type classes that contain associated types. Here's why that is. There is no fundamental obstacle to translating associated type synonyms into Coq; associated type synonyms make type classes records of types and values instead of just records of values, which is not a distinction Coq makes. However, this CPSed encoding breaks down. Consider the following Haskell type class:

```
class D a where
  type T a
  g :: T a -> a
```

Translating this to the CPSed form would be a problem. First, we'd get

```
Inductive D__Dict a := D__Dict_Build {
  T__ : Type ;
  g__ : T__ -> a }.

Definition D a :=
  forall r__, (D__Dict a -> r__) -> r__.
Existing Class D.
```

So far, so good. But now we have to encode `T` and `g`, and we run into problems. `T` goes smoothly:

```
Definition T a `{g__0__ : D a} : Type :=
```

```
  g__0__ _ (T__ a).
```

But defining `g` won't work. We define it by "unwrapping" the CPS-ed `D` at the type of the type class method, which has above been inferred (it's the `_` in every `g__0__`). But here, the type depends on the `D__Dict a` argument, since it contains the definition of `T` and thereby determines the value of `T a`! So this encoding scheme won't work.

Thus, enter the **`simple class`** edit: it replaces this complex desugaring with a direct one in terms of Coq type classes. Because the desugaring affects only definitions of type classes and the auto-generated instance machinery, it does not change use-sites and so can be switched around without clients having to care.

Examples.
Input:

```
# Edits
simple class Mod.C
simple class Mod.D

-- Haskell
{-# LANGUAGE TypeFamilies #-}

module Mod where

class C a where
  m :: a
  f :: a -> Bool

instance C () where
  m = ()
  f _ = True

class D a where
  type T a
  g :: T a -> a

instance D () where
  type T () = Bool
  g _ = ()
```

Output:

```
(* Coq *)

(* Converted type declarations: *)

Class D a := {
```

```
  T : Type ;
  g : T -> a }.

Arguments T _ {_}.

Class C a := {
  f : a -> bool ;
  m : a }.

(* Converted value declarations: *)

Local Definition C__unit_f : unit -> bool :=
  fun arg_0__ => true.

Local Definition C__unit_m : unit :=
  tt.

Program Instance C__unit : C unit :=
  { f := C__unit_f ; m := C__unit_m }.

Local Definition D__unit_T : Type :=
  bool.

Local Definition D__unit_g : D__unit_T -> unit :=
  fun arg_0__ => tt.

Program Instance D__unit : D unit :=
  { T := D__unit_T ; g := D__unit_g }.
```

8.5.3. inline mutual.

Edit: `inline mutual` *Qualid*

Effect: (Originally from the documentation.[39] The example was initially presented on Stack Overflow.[40]) The specified definition must be part of a mutually recursive set of definitions. Instead of being defined as another mutual fixpoint, it will be inlined into each of the other mutual fixpoints that needs it with a `let`-binding; additionally, a top-level Coq definition is generated for each `let`-bound function that simply calls into the predefined recursive functions.

This facility is useful when translating groups of mutually recursive functions that contain "preprocessing" or "postprocessing" functions, where the group is otherwise

[39] Available at https://hs-to-coq.readthedocs.io/en/latest/edits.html#inline-mutual-move-mutually-recursive-functions-into-let-bindings

[40] https://stackoverflow.com/q/52599324/237428

structurally recursive. These functions are not "truly" mutual recursive, as they just hand along values of the type being recursed, and so if Coq could only see through them, everything would work fine. And indeed, as `let`-bindings, Coq can see through them.

As an example, consider the following pair of mutually recursive data types, which represent a `Forest` of nonempty `Tree`s. Each `Branch` of a `Tree` holds an extra boolean flag, which we can extract with `isOK`. In Haskell:

```
data Forest a = Empty
              | WithTree (Tree a) (Forest a)

data Tree a = Branch Bool a (Forest a)

isOK :: Tree a -> Bool
isOK (Branch ok _ _) = ok
```

And in cleaned-up Coq:

```
Inductive Forest a : Type
  := Empty    : Forest a
  |  WithTree : Tree a -> Forest a -> Forest a
with Tree a : Type
  := Branch : bool -> a -> Forest a -> Tree a.

Arguments Empty    {_}.
Arguments WithTree {_} _ _.
Arguments Branch   {_} _ _ _.

Definition isOK {a} : Tree a -> bool :=
  fun '(Branch ok _ _) => ok.
```

Now we can define a pair of mapping functions that only apply a function inside subtrees where the boolean flag is true. The Haskell code is simple:

```
mapForest :: (a -> a) -> Forest a -> Forest a
mapForest f Empty           = Empty
mapForest f (WithTree t ts) = WithTree (mapTree   f t)
                                       (mapForest f ts)

mapTree :: (a -> a) -> Tree a -> Tree a
mapTree f t | isOK t    = mapOKTree f t
            | otherwise = t

mapOKTree :: (a -> a) -> Tree a -> Tree a
mapOKTree f (Branch ok x ts) = Branch ok (f x) (mapForest f ts)
```

However, the (cleaned-up) Coq translation fails:

```
Fail Fixpoint mapForest
    {a} (f : a -> a) (ts0 : Forest a) {struct ts0} : Forest a :=
  match ts0 with
  | Empty         => Empty
  | WithTree t ts => WithTree (mapTree f t) (mapForest f ts)
  end
with mapTree {a} (f : a -> a) (t : Tree a) {struct t} : Tree a :=
  if isOK t
  then mapOKTree f t
  else t
with mapOKTree {a} (f : a -> a) (t : Tree a) {struct t} : Tree a :=
  match t with
  | Branch ok x ts => Branch ok (f x) (mapForest f ts)
  end.
```

The issue is that `mapTree` calls `mapOKTree` on the *same* term, and not a subterm. But this just a preprocessing/postprocessing split – there's nothing *actually* recursive going on.

But with

```
inline mutual mapOKTree
```

we instead get working Coq code (again, cleaned up):

```
Fixpoint mapForest {a} (f : a -> a) (ts0 : Forest a) {struct ts0}
           : Forest a :=
  match ts0 with
  | Empty         => Empty
  | WithTree t ts => WithTree (mapTree f t) (mapForest f ts)
  end
with mapTree {a} (f : a -> a) (t : Tree a) {struct t} : Tree a :=
  let mapOKTree {a} (f : a -> a) (t : Tree a) : Tree a :=
        match t with
        | Branch ok x ts => Branch ok (f x) (mapForest f ts)
        end in
  if isOK t
  then mapOKTree f t
  else t.

Definition mapOKTree {a} (f : a -> a) (t : Tree a) : Tree a :=
  match t with
  | Branch ok x ts => Branch ok (f x) (mapForest f ts)
  end.
```

This is the idea. However, to be completely fair, we never produce **Fixpoint** commands; both in the failing case and in the successful case, we generate fix terms. In this example, this looks like (reindented)

```
Definition mapForest {a} : (a -> a) -> Forest a -> Forest a :=
  fix mapTree f t :=
    let mapOKTree arg_0__ arg_1__ :=
          match arg_0__, arg_1__ with
          | f, Branch ok x ts => Branch ok (f x) (mapForest f ts)
          end in
    if isOK t : bool
    then mapOKTree f t
    else t
  with mapForest arg_0__ arg_1__ :=
    match arg_0__, arg_1__ with
    | f, Empty => Empty
    | f, WithTree t ts => WithTree (mapTree f t) (mapForest f ts)
    end
  for mapForest.

Definition mapOKTree {a} : (a -> a) -> Tree a -> Tree a :=
  fun arg_0__ arg_1__ =>
    match arg_0__, arg_1__ with
    | f, Branch ok x ts => Branch ok (f x) (mapForest f ts)
    end.

Definition mapTree {a} : (a -> a) -> Tree a -> Tree a :=
  fix mapTree f t :=
    let mapOKTree arg_0__ arg_1__ :=
          match arg_0__, arg_1__ with
          | f, Branch ok x ts => Branch ok (f x) (mapForest f ts)
          end in
    if isOK t : bool
    then mapOKTree f t
    else t
  with mapForest arg_0__ arg_1__ :=
    match arg_0__, arg_1__ with
    | f, Empty => Empty
    | f, WithTree t ts => WithTree (mapTree f t) (mapForest f ts)
    end
  for mapTree.
```

Examples. To see **inline mutual** in action, see the mapOKTree example above.

8.6. Rewriting expressions

8.6.1. rewrite.

Edit: `rewrite forall Word*, Term= = Term`

Effect: The *rewrite* edit replaces arbitrary expressions in the translated Coq code. Everywhere that the the expression before the equal sign is found,[41] it is replaced by the expression on the right. The `forall` clause brings into scope metavariables; everywhere a metavariable shows up on the left-hand side, it can match an arbitrary expression, and that expression is substituted in for the metavariable on the right-hand side during replacement. Variables that were not brought into scope are matched literally, and need to be fully module-qualified if necessary.

This edit often wants to be used with the `in` meta-edit (Section 8.9.1) to scope its effects to within a specific definition. However, this isn't mandatory; sometimes, you are expressing global rules about code structure, and a global rewrite is appropriate.

Examples.
Input:

```
# Edits

# Strictness does not affect the translation
rewrite forall a b, GHC.Prim.seq a b = b

# Remove an inessential use of laziness
rewrite forall xs, ↩
  GHC.List.zip (GHC.Enum.enumFrom (GHC.Num.fromInteger 0)) xs ↩
  = GHC.List.zip (GHC.Enum.enumFromTo ↩
                    (GHC.Num.fromInteger 0) ↩
                    (GHC.Num.fromInteger (GHC.List.length xs))) ↩
                 xs


-- Haskell
strictAppend :: [a] -> [a] -> [a]
strictAppend xs ys = xs `seq` ys `seq` (xs ++ ys)

evens :: [a] -> [a]
evens xs = map snd . filter (even . fst) $ zip [0..] xs
```

[41] *Term=* refers to a term that does not contain a bare `=`, as otherwise there would be ambiguity.

Output:

```
(* Coq (reformatted) *)
Definition strictAppend {a} : list a -> list a -> list a :=
  fun xs ys => Coq.Init.Datatypes.app xs ys.

Definition evens {a} : list a -> list a :=
  fun xs =>
    (GHC.Base.map Data.Tuple.snd GHC.Base.∘
     GHC.List.filter (GHC.Real.even GHC.Base.∘ Data.Tuple.fst))
      (GHC.List.zip (GHC.Enum.enumFromTo
                       #0 (GHC.Num.fromInteger (GHC.List.length xs)))
                    xs).
```

8.6.2. rename.

Edit: `rename` *Namespace Qualid = Qualid*

Effect: The `rename` edit replaces one Haskell name with another. The *Namespace* can be `type` or `value`, corresponding to Haskell's two namespaces. Suppose we have the Haskell file

```
module Mod where

data X = X

idX :: X -> X
idX X = X
```

which defines and uses a trivial unit type whose type and constructor have the same name, `X`. Then the edit `rename type Mod.X = Mod.T` changes only the type name, producing the output

```
Inductive T : Type := | X : T.

Definition idX : T -> T :=
  fun '(X) => X.
```

On the other hand, the edit `rename value Mod.X = Mod.V` instead changes the name of the constructor, producing the output

```
Inductive X : Type := | V : X.

Definition idX : X -> X :=
  fun '(V) => V.
```

This also demonstrates one reason why `rename` is important: without either of those edits, the translated definition of the type would be `Inductive X : Type := | X : X.`, which is illegal as `X` is being used in two different ways and Coq only has one namespace.

The customary pattern is to rename constructors that need depunning to begin with `Mk_`.

The `rename` edit can also help avoid names that are already in use (reserved names are avoided automatically by `hs-to-coq`). Lastly, it can be used to effectively redefine types to be existing Coq types, by renaming all uses to refer to the Coq standard library (or similar) and then skipping the original definition; this is how we link things like Haskell `[]` and `Bool` to Coq `list` and `bool`.

Examples.
Input:

```
# Edits
rename type  GHC.Types.[] = list
rename value GHC.Types.[] = nil
rename value GHC.Types.:  = cons
rename value GHC.Base.++  = Coq.Init.Datatypes.app

rename type  GHC.Base.Maybe   = option
rename value GHC.Base.Just    = Some
rename value GHC.Base.Nothing = None

rename value Data.Functor.Identity.Identity = ↩
             Data.Functor.Identity.Mk_Identity

-- Haskell
module Mod where

import Data.Functor.Identity
-- newtype Identity a = Identity { runIdentity :: a }

f :: Identity a -> [Maybe a]
f (Identity x) = [Just x, Nothing]
```

Output:

```
(* Coq *)
Definition f {a}
    : Data.Functor.Identity.Identity a -> list (option a) :=
  fun '(Data.Functor.Identity.Mk_Identity x) =>
    cons (Some x) (cons None nil).
```

8.6.3. rename module.

Edit: `rename module Word Word`

Effect: The *rename module* edit replaces the name of one Haskell module with another. This is much like the above **rename** edit, with three key differences:

(1) It affects modules instead of types and values; thus it affects portions of names, not whole names.
(2) It affects the output filename, since the Coq module `Mod` lives in the file `Mod.v`.
(3) Two different Haskell modules can be renamed into a single Coq module, which will cause `hs-to-coq` to merge the contents of those two Haskell modules and treat them like a single module during compilation. This is particularly useful when translating mutually recursive Haskell modules, which are used in GHC; we combine these into a single mega-module.

Examples.
Input:

```
# Edits
rename module FirstHalf  Whole
rename module SecondHalf Whole

rename module HsMod CoqMod

-- Haskell

-- FirstHalf.hs
module FirstHalf where

true :: Bool
true = True

-- SecondHalf.hs
module SecondHalf where

false :: Bool
false = False

-- HsMod.hs
module HsMod where

import FirstHalf
import SecondHalf

booleans :: [Bool]
booleans = [true, false]
```

Output:

```
(* Coq *)

(* Whole.v *)

Definition true : bool :=
  true.

Definition false : bool :=
  false.

(* CoqMod.v *)

(* Converted imports: *)

Require Whole.

(* Converted value declarations: *)

Definition booleans : list bool :=
  cons Whole.true (cons Whole.false nil).
```

8.7. Providing extra information

8.7.1. data kinds.

Edit: `data kinds` *Qualid Term* $\overline{\text{, Term}}$

Effect: The `data kinds` edit allows the `hs-to-coq` user to provide kind annotations (which are really just type annotations) on data type definitions that the original Haskell program omitted. For the most part, Haskell programmers don't write kind annotations, since GHC can infer the correct kinds; because kinds are usually simple, Coq can also infer the correct kinds in most cases. But not always. The problem arises when data types have parameters that are used in slightly unusual ways: higher-order, or phantom, or polymorphic. In these cases, GHC and Coq will sometimes disagree about the correct kind for a parameter, or even whether or not one can be inferred. Enter `data kinds`, which for a data type with n parameters specifies $n + 1$ kinds: one for each of the type variables, and one for the return kind.

Examples.
Input:

```
# Edits
data kinds Mod.Phantom Type, Type
data kinds Mod.HO (Type -> Type), Type, Type
```

```
-- Haskell
module Mod where

-- Coq would fail to deduce the type of either "a"
data Phantom a = MkPhantom
data HO f a = MkHO (f a)
```

Output:

```
(* Coq *)
Inductive Phantom (a : Type) : Type := | MkPhantom : Phantom a.

Inductive HO (f : Type -> Type) (a : Type) : Type :=
| MkHO : (f a) -> HO f a.
```

8.7.2. class kinds.

Edit: `class kinds` *Qualid Term* $\overline{\texttt{, } \textit{Term}}$

Effect: Data type definitions are not the only source of type-level definitions in Haskell; there are also type classes. The `class kinds` edit is like `data kinds`, but for type classes; it allows the `hs-to-coq` user to provide kind annotations (which are still really just type annotations) on *class* definitions that the original Haskell program omitted. Again, these are usually omitted and inferred identically by both GHC and Coq, but not always, particularly in the case of higher-order uses.

Examples.

Input:

```
# Edits
class kinds Mod.Alternative Type -> Type, Type

-- Haskell
module Mod where

-- It's clear to Coq that @f : _ -> Type@, but not what @_@ is
class Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Output:

```
(* Coq *)
Record Alternative__Dict (f : Type -> Type) :=
  Alternative__Dict_Build {
    empty__ : forall {a}, f a ;
    op_zlzbzg____ : forall {a}, f a -> f a -> f a }.

Definition Alternative (f : Type -> Type) :=
```

```
  forall r__, (Alternative__Dict f -> r__) -> r__.
Existing Class Alternative.

Definition empty `{g__0__ : Alternative f} : forall {a}, f a :=
  g__0__ _ (empty__ f).

Definition op_zlzbzg__ `{g__0__ : Alternative f}
   : forall {a}, f a -> f a -> f a :=
  g__0__ _ (op_zlzbzg____ f).

Notation "'_<|>_'" := (op_zlzbzg__).

Infix "<|>" := (_<|>_) (at level 99).

(* Converted value declarations: *)

Module Notations.
Notation "'_Mod.<|>_'" := (op_zlzbzg__).
Infix "Mod.<|>" := (_<|>_) (at level 99).
End Notations.
```

8.7.3. delete unused type variables. (Originally from the documentation.[42])

Edit: `delete unused type variables` *Qualid*

Effect: Don't translate binders for any type variables that aren't visibly used in the specified definition.

An explanation: sometimes, poly-kinded Haskell data types have extra invisible type parameters. For instance, in `Data.Functor.Const`, we have the type

```
newtype Const a b = Const { getConst :: a }
```

which, since the `PolyKinds` language extension is enabled at its definition site, is secretly

```
newtype Const {k} (a :: Type) (b :: k) = Const { getConst :: a }
```

Often, such as here, this doesn't show up in the translated Coq code; we get

```
Inductive Const a b : Type := Mk_Const (getConst : a) : Const a b.
```

Unlike in Haskell 2010, `b` causes an inference failure error instead of being inferred to have kind `Type`; we have to use **`data kinds`** (Section 8.7.1) to fix that. But that's not the case at every use of this data type; sometimes, we still introduce spurious kind variables in the translation. For example, the derived `Eq` instance for `Const` is translated to

[42] Available at https://hs-to-coq.readthedocs.io/en/latest/edits.html#delete-unused-type-variables-remove-unused-type-variables-from-a-declaration

```
Program Instance Eq___Const {a} {k} {b} `{GHC.Base.Eq_ a}
   : GHC.Base.Eq_ (Const a b : GHC.Prim.TYPE GHC.Types.LiftedRep) :=
  fun _ k =>
    k {| GHC.Base.op_zeze____ := Eq___Const_op_zeze__ ;
         GHC.Base.op_zsze____ := Eq___Const_op_zsze__ |}.
```

The implicit argument `{k}` isn't useful in the Coq code, and causes a type-checking failure when its type cannot be determined. We can avoid this with

```
delete unused type variables Data.Functor.Const.Eq___Const
```

which will detect that `k` is not referenced in the body of the definition, drop the `{k}` binder, and leave the definition with just the `{a}` and `{b}` binders it needs:

```
Program Instance Eq___Const {a} {b} `{GHC.Base.Eq_ a}
   : GHC.Base.Eq_ (Const a b : GHC.Prim.TYPE GHC.Types.LiftedRep) :=
  fun _ k =>
    k {| GHC.Base.op_zeze____ := Eq___Const_op_zeze__ ;
         GHC.Base.op_zsze____ := Eq___Const_op_zsze__ |}.
```

Examples.

Input:

```
# Edits
# Adapted from our translation of `base`
rename value Data.Functor.Const.Const = Data.Functor.Const.Mk_Const
data kinds Data.Functor.Const.Const Type, Type, Type
delete unused type variables Data.Functor.Const.Eq___Const

-- Haskell
-- Simplified from `base`
{-# LANGUAGE PolyKinds #-}
module Mod where

newtype Const a b = Const { getConst :: a } deriving Eq
```

Output:

```
(* Coq *)

Inductive Const (a : Type) (b : Type) : Type
  := | Mk_Const (getConst : a) : Const a b.

Program Instance Eq___Const {a} {k} {b} `{GHC.Base.Eq_ a}
   : GHC.Base.Eq_ (Const a b : GHC.Prim.TYPE GHC.Types.LiftedRep) :=
  fun _ k__ =>
    k__ {| GHC.Base.op_zeze____ := Eq___Const_op_zeze__ ;
           GHC.Base.op_zsze____ := Eq___Const_op_zsze__ |}.
```

8.7.4. order.

Edit: `order` *Qualid+*

Effect: Adjusts the order in which definitions are output after translation. For example, the edit `order Mod.f1 Mod.f2 Mod.f3` will ensure that, in `mod.v`, the definition of `f1` comes before the definition of `f2`, which itself comes before the definition of `f3`. Other definitions may intervene, but this order will be maintained.

By default, `hs-to-coq` topologically sorts all definitions in a module (or a `let` expression, or something similar) before writing them out. While Haskell does not care about definition order, allowing names within a file (or a a`let` expression, or something similar) to find their referent either before *or after* their use, Coq is not so lenient, and requires all references to names to occur to syntactically earlier binding sites.

However, while this works most of the time, a simple syntactically-driven topological sort is not enough for all cases of interest. Thanks to type classes, names may be referenced invisibly; a dependency on the `Num Int` type class instance is not visible in the source in Haskell or Coq. Thus, the `order` edit, which allows the programmer to specify the dependencies that `hs-to-coq` cannot see. This is particularly useful when defining type class methods that may depend on other type class instances; these often feature these invisible name references in a way that `hs-to-coq` gets wrong.

Examples.
Input:

```
# Edits
# Ensures the `C (option a)` type class instance is in scope before
# the definition of `m` for the `C (list a)` instance, despite it
# not appearing in the source code.
order Mod.C__option Mod.C__list_m
simple class Mod.C # For concision

-- Haskell
module Mod where

import Data.Maybe

class C a where
  m :: a

instance C a => C (Maybe a) where
  m = Just m

instance C a => C [a] where
  m = maybeToList m
```

Output:

```
(* Coq *)

(* Converted type declarations: *)

Class C a := {
  m : a }.

(* Converted value declarations: *)

Local Definition C__option_m {inst_a} `{C inst_a}
    : (option inst_a) :=
  Some m.

Program Instance C__option {a} `{C a} : C (option a) :=
  { m := C__option_m }.

Local Definition C__list_m {inst_a} `{C inst_a} : list inst_a :=
  Data.Maybe.maybeToList m.

Program Instance C__list {a} `{C a} : C (list a) :=
  { m := C__list_m }.
```

8.7.5. manual notation.

Edit: `manual notation` *Word*

Effect: This edit allows the user to specify that a module has Coq `Notation`s defined in preamble files, allowing them to work smoothly with `hs-to-coq`'s automatic notation translation. (See Section 3.5 for the details of that translation.) The idea behind `manual notation` is that you may need to define some of our faux-qualified operators – or other such operators you want to expose – by hand, in a preamble or midamble Coq file. But these then won't be automatically imported through the `Notation` module, and will only be available if the module is actually imported, not simply required. This is what `manual notation` allows you to change: `manual notation Some.Mod` simply places the line `Export ManualNotations.` at the top of the `Notation` module (and assures that the `Notation` module exists). This means that whenever `Some.Mod.Notation` is manually imported by a downstream module, anything from the `ManualNotation` module is made available to that downstream module as well. This edit does nothing to *generate* that module; it is designed for uses where you've already defined such notations and placed them in there, hence the need for a preamble or midamble. It is strictly for making the code therein more available to `hs-to-coq`'s internal tracking.

We need to use this edit exactly once, for the equality and comparison type class operators in `GHC.Base`.

Examples.

Input:

```
# Edits
manual notation Operators
skip Operators.!

-- Haskell
-- Operators.hs
module Operators where

(&) :: a -> (a -> b) -> b
x & f = f x

(!) :: Int -> Int -> Int
(!) = somethingLowLevel

-- Mod.hs
module Mod where

import Operators

x :: Int
x = (1 ! 2) & negate

(* Coq *)
(* preamble.v *)
Definition bang := plus.
Notation "'_!_'" := bang.
Infix "!" := _!_ (at level 99).

Module ManualNotations.
Notation "'_Operators.!_'" := bang.
Infix "Operators.!" := _!_ (at level 99).
End ManualNotations.
```

Output:

```
(* Coq *)

(* Operators.v  *)

(* Preamble *)
```

```
Definition bang := plus.
Notation "'_!_'" := bang.
Infix "!" := _!_ (at level 99).

Module ManualNotations.
Notation "'_Operators.!_'" := bang.
Infix "Operators.!" := _!_ (at level 99).
End ManualNotations.

(* No imports to convert. *)

(* No type declarations to convert. *)

(* Converted value declarations: *)

Definition op_za__ {a} {b} : a -> (a -> b) -> b :=
  fun x f => f x.

Notation "'_&_'" := (op_za__).

Infix "&" := (_&_) (at level 99).

Module Notations.
Export ManualNotations.
Notation "'_Operators.&_'" := (op_za__).
Infix "Operators.&" := (_&_) (at level 99).
End Notations.

(* Mod.v *)

(* Converted imports: *)

Require GHC.Num.
Require Operators.
Import GHC.Num.Notations.
Import Operators.Notations.

(* Converted value declarations: *)

Definition x : GHC.Num.Int :=
  (#1 Operators.! #2) Operators.& GHC.Num.negate.
```

8.7.6. set type.

Edit: `set type` *Qualid* `:` *Term*
`set type` *Qualid* `no type`

Effect: This edit changes the type annotation `hs-to-coq` places on a term. By default, `hs-to-coq` uses the type annotation provided in the Haskell source code, or none if no such type annotation was specified. The `set type` edit allows the user to override this, either by using the specified type annotation (in the `: Term` case) or by omitting a type annotation (in the `no type` case). One particularly important use of this edit is to add `Default` constraints to allow for the use of "nontermination" (see Section 7.4). It is pleasant to note that this edit can never introduce unsoundness; because Coq typechecks the resulting definition, the only harm that can come from this edit is that a type could be too restrictive.

Examples.
Input:

```
# Edits
set type Mod.head : forall {a} `{(GHC.Err.Default a)}, list a -> a
set type Mod.unit no type

-- Haskell
module Mod where

head :: [a] -> a
head (x:xs) = x

unit :: ()
unit = ()
```

Output:

```
(* Coq *)

Definition unit :=
  tt.

Definition head {a} `{GHC.Err.Default a} : list a -> a :=
  fun arg_0__ =>
    match arg_0__ with
    | cons x xs => x
    | _ => GHC.Err.patternFailure
    end.
```

8.7.7. data type arguments.

Edit: `data type arguments` *Qualid* (`parameters` : *Qualid**)? (`indices` : *Qualid**)?

Effect: In Haskell, data types do not differentiate between their parameters and their indices, either syntactically or semantically. In Coq, indices are distinguished both syntactically (parameters are bound as arguments to the type, indices are built through the arrows on its type/universe annotation) and semantically (parameters come first and don't raise the universe level, indices come last and do raise the universe level). Thus, a Haskell GADT translated naïvely into Coq may be invalid for this reason (if the GADT does separate its indices from its parameter,s then everything will work fine). The `data type arguments` edit, given the name of a data type, reorganizes its Haskell parameters into Coq parameters and indices, adjusts the arguments to the translated type appropriately. The names used in the parameter and index lists must be exactly the same as the names bound in the Haskell file; this is only about reorganization, not rewriting. Furthermore, changing the order of the type arguments likely will not do what you want, as it will not reorder the arguments at use sites.

Examples.
Input:

```
# Edits
data type arguments Mod.Has (parameters : a) (indices : p)
data kinds Mod.Has Type, YN

-- Haskell
{-# LANGUAGE GADTs, DataKinds #-}

module Mod where

data YN = Y | N

data Has a p where
  Yes :: a -> Has a Y
  No  :: Has a N

maybeHas :: Has a p -> Maybe a
maybeHas (Yes x) = Just x
maybeHas No      = Nothing
```

Output:

```
(* Coq *)

(* Converted type declarations: *)
```

```
Inductive YN : Type := | Y : YN |  N : YN.

Inductive Has (a : Type) : forall (p : YN), Type
  := | Yes : forall {p : YN}, a -> Has a Y
  |  No : forall {p : YN}, Has a N.

(* Converted value declarations: *)

Definition maybeHas {a} {p} : Has a p -> option a :=
  fun arg_0__ =>
    match arg_0__ with
    | Yes x => Some x
    | No => None
    end.
```

8.8. Proving termination

8.8.1. termination.

Edit: `termination` *Qualid TerminationArgument*

Effect: This edit allows us to avoid running head-on into Coq's termination checker by specifying an alternative proof that the specified function is terminating. In many ways, the four distinct termination arguments are three distinct edits, but they all serve the same purpose: alter the translation of a Coq `fix`point so that you can prove that it will terminate. The four termination arguments are:[43]

(1) `{ measure` *Atom* $\overline{\texttt{(}\ \textit{Term}\ \texttt{)}}^{?}$ `}`
(2) `{ wf` *Atom Qualid* `}`
(3) `deferred`
(4) `corecursive`

The first two use Coq's **`Program`** command to translate the fixpoint (Sozeau, 2006); **`deferred`** uses a highly classical fixpoint axiom for maximum flexibility, requiring the user to provide a termination argument when calling the function (Breitner et al., 2018); and **`corecursive`** uses (guarded) corecursion instead of (structural) recursion.

When using one of the termination arguments that invokes **`Program`**, the `measure` and `wf` arguments are passed on to the **`Program Fixpoint`** that is now being used to define `f`. The former, **`termination`** `f {measure (m x)}`, permits the user to prove that `f` is terminating by proving that `m x` is monotonically decreasing on ever call (by default, with respect to `lt`, the less than relation on natural numbers, but another relation can be provided). The latter, **`termination`** `f {wf R x}`, is the same as **`termination`** `f {measure x R}`; in other words, these two are same, but they take their parameters in opposite orders and only `measure` permits you to omit one parameter. These annotations are documented further in the Coq manual, which

[43] `Atom` denotes either a parenthesized term or a single term that can be written without a space (usually an identifier).

explains the use of `Program` (The Coq Development Team, 2020b, "Program"[44]). The use of `Program` incurs immediate proof obligations; to solve these, see the next edit, `obligations`.

When using the termination argument `deferred`, the fixpoint is translated to a call to our `GHC.DeferredFix.deferredFix` axiom:

```
Axiom deferredFix:
  forall {a r} `{Default r}, ((a -> r) -> (a -> r)) -> a -> r.
```

This axiom has the same shape as any classic fixpoint axiom that is limited to producing functions, except that it constraints the result type of the function to be inhabited through the use of the `Default r` constraint. This allows us to ensure that `deferredFix` is consistent, as we can never use it to produce `False`. However, the evaluation rule for `deferredFix` is highly classical:

```
Definition recurses_on {a b}
                       (P : a -> Prop) (R : a -> a -> Prop)
                       (f : (a -> b) -> (a -> b)) :=
  forall g h x,
    P x ->
    (forall y, P y ->  R y x -> g y = h y) ->
    f g x = f h x.

Axiom deferredFix_eq_on:
  forall {a b} `{Default b}
         (f : (a -> b) -> (a -> b))
         (P : a -> Prop)
         (R : a -> a -> Prop),
    well_founded R -> recurses_on P R f ->
    forall x, P x -> deferredFix f x = f (deferredFix f) x.
```

The statement `recurses_on P R f` says, informally, that any recursive calls in `f` are on arguments that are smaller with respect to `R`; we use `P` to restrict the domain, so we can apply the axiom in more cases. Then as long as this holds and `R` is well-founded, we can unroll `deferredFix` one step (Breitner et al., 2018, §5.4). Because all we need to use `deferredFix` is a proof that the result type is inhabited, we can use `termination f deferred` even without a proof of termination (we are deferring that proof, hence the name). This is helpful, as it allows further separation of translation and proof; the sheer classical power of the axiom also allows us to employ a wide variety of termination arguments.

Finally, when using the termination argument `corecursion`, the `fix` is simply translated to a `cofix`. No other work is done, so the cofixpoint must be guarded, just as fixpoints must be structural.

Examples.

[44]This chapter is available at https://coq.inria.fr/refman/addendum/program.html.

Input:

```
# Edits
termination TerminationProgram.smaller {measure (length arg_0__)}
obligations TerminationProgram.smaller ↩
  intros; Tactics.program_simpl; ↩
  do 3 try destruct arg_0__; simpl; intuition

-- Haskell
module TerminationProgram where

-- A very fancy way to compute []
smaller :: [Int] -> [Int]
smaller [] = []
smaller xs = smaller (drop 2 xs)
```

Output:

```
(* Coq *)

Program Fixpoint smaller (arg_0__ : list GHC.Num.Int)
                         {measure (length arg_0__)}
                       : list GHC.Num.Int
                      := match arg_0__ with
                         | nil => nil
                         | xs => smaller (GHC.List.drop #2 xs)
                         end.
Solve Obligations with
  (intros; Tactics.program_simpl;
   do 3 try destruct arg_0__; simpl; intuition).
```

Input:

```
# Edits
rename type GHC.Integer.Type.Integer = GHC.Num.Integer
termination TerminationDeferred.ackermann deferred

-- Haskell
module TerminationDeferred where

-- The Ackermann-Péter function, from Wikipedia
-- <https://en.wikipedia.org/wiki/Ackermann_function>
ackermann :: Integer -> Integer -> Integer
ackermann m n
  | m <= 0 = n+1
  | n <= 0 = ackermann (m-1) 1
```

```
  | otherwise = ackermann (m-1) (ackermann m (n-1))
```

Output:

```
(* Coq *)

Definition ackermann
    : GHC.Num.Integer -> GHC.Num.Integer -> GHC.Num.Integer :=
  GHC.DeferredFix.deferredFix1
    (fun ackermann (m n : GHC.Num.Integer) =>
       if m GHC.Base.<= #0 : bool
       then n GHC.Num.+ #1
       else if n GHC.Base.<= #0 : bool
       then ackermann (m GHC.Num.- #1) #1
       else ackermann (m GHC.Num.- #1)
                      (ackermann m (n GHC.Num.- #1))).
```

Input:

```
# Edits
coinductive TerminationCorecursive.Stream
termination TerminationCorecursive.repeat corecursive

-- Haskell
module TerminationCorecursive where

data Stream a = Cons a (Stream a)

repeat :: a -> Stream a
repeat x = Cons x (repeat x)
```

Output:

```
(* Coq *)

(* Converted type declarations: *)

CoInductive Stream a : Type := | Cons : a -> (Stream a) -> Stream a.

(* Converted value declarations: *)

Definition repeat {a} : a -> Stream a :=
  cofix repeat (x : a) := Cons x (repeat x).
```

8.8.2. obligations.

Edit: `obligations` *`Qualid TrivialLtac`*

Effect: The `obligations` edit takes the definition with the given name, translates it using the `Program` command (Sozeau, 2006), and then uses the given tactic to solve the generated obligations. This is commonly paired with the `termination` edit using the `{measure e R}` or `{wf R e}` termination arguments, which will have already caused the definition to be translated with `Program`; however, this isn't necessary, in case the programmer wants the facilities of `Program` simply for dealing with implicit arguments.

When `Program` is used, proof obligations are incurred; in interactive development, it's typical to go through these one by one with `Next Obligation.`, which places you in the appropriate proof environment after running the tactic `Coq.ProgramTactics.program_simpl`. However, it's not uncommon to instead use `Solve Obligations`, which takes a single Ltac tactic and applies it to solve every single obligation in one fell swoop. This is much more suitable to `hs-to-coq`'s automatic translation, and `obligations f tactic` will follow the definition of `f` with `Solve Obligations (tactic).` It is common to begin the tactic with `Tactics.program_simpl; ...`, as this brings things into the (more useful) state they would be in after a `Next Obligation.` command.

The grammar production *TrivialLtac* indicates that while we are still (Section 8.4.1) not parsing Ltac, we have to do a little work; we support parsing (nested) applications of identifiers, numbers, underscores, `@ident` terms, and parenthesized terms, separated by `;` or `||`. Anything more complex must be defined as an actual tactic in the preamble or midamble, and then referenced.

Finally, there is one special case: writing `obligations f admit` generates `Admit Obligations.` instead of `Solve Obligations with (admit).`, as the latter does not work properly.

Examples. (From Section 1.1.)
Input:

```
# Edits
termination Sort.merge {measure (length arg_0__ + length arg_1__)}
obligations Sort.merge Tactics.program_simpl; simpl; lia

-- Haskell
module Sort where

merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y    = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys

(* Coq *)
(* preamble.v *)
```

```
Require Import Lia.
```

Output:

```
(* Coq *)

(* Preamble *)

Require Import Lia. (* For the `lia` tactic *)

(* Converted imports: *)

Require GHC.Base.
Import GHC.Base.Notations.

(* No type declarations to convert. *)

(* Converted value declarations: *)

Program Fixpoint merge
    {a} `{Ord a} (arg_0__ arg_1__ : list a)
    {measure (length arg_0__ + length arg_1__)} : list a :=
  := match arg_0__, arg_1__ with
     | nil, ys => ys
     | xs, nil => xs
     | cons x xs, cons y ys =>
         if Bool.Sumbool.sumbool_of_bool (x GHC.Base.<= y)
         then cons x (merge xs (cons y ys)) else
         cons y (merge (cons x xs) ys)
     end.
Solve Obligations with (Tactics.program_simpl; simpl; lia).
```

8.8.3. coinductive.

Edit: `coinductive` *`Qualid`*

Effect: Given a data type definition named *`Qualid`*, the `coinductive` edit instructs `hs-to-coq` to translate that definition as a `CoInductive` instead of an `Inductive`. No other changes are made; any uses of this type must be correctly coinductive, or made such with edits such as `termination f corecursive`. Even with this edit, a single type cannot be used both inductively and coinductively; although Haskell merges lists and streams, Coq cannot.

Examples.

Input:

```
# Edits
coinductive Mod.Stream
coinductive Mod.CoList

-- Haskell
module Mod where

data Stream a = SCons a (Stream a)

data CoList a = CoNil
              | CoCons a (CoList a)
```

Output:

```
(* Coq *)
CoInductive Stream a : Type
  := | SCons : a -> (Stream a) -> Stream a.

CoInductive CoList a : Type
  := | CoNil : CoList a
  |  CoCons : a -> (CoList a) -> CoList a.
```

8.9. Meta-edits

8.9.1. in.

Edit: `in` *`Qualid Edit`*

Effect: The unique meta-edit, `in` allows a whole edit – any of the previous forms we've discussed, or another nested `in` – to only apply within the definition of the name *`Qualid`*. This name can be any sort of top-level definition (variable, type class, type class method, etc.), as well as a `let`-bound local variable (likely only applicable within an outer `in` that references a top-level name). This is useful particularly for edits such as **`rewrite`** (Section 8.6.1) or **`skip case pattern`** (Section 8.2.6) that have broad effects only useful in narrow areas, but can be applied to anything.

Examples.
Input:

```
# Edits
rewrite forall a, GHC.Num.fromInteger 1 GHC.Num.* a = a
in Mod.double2 rewrite forall a, ↩
  a GHC.Num.+ a = (GHC.Num.fromInteger 2) GHC.Num.* a
in Mod.quadruple rewrite forall a, ↩
  a GHC.Num.+ a = (GHC.Num.fromInteger 2) GHC.Num.* a
```

```
-- Haskell
module Mod where

double1 :: Int -> Int
double1 x = 1*x + 1*x

double2 :: Int -> Int
double2 x = 1*x + 1*x

quadruple :: Int -> Int
quadruple x = let x' = 1*x + 1*x in 1*x' + 1*x'
```

Output:

```
(* Coq *)
Definition quadruple : GHC.Num.Int -> GHC.Num.Int :=
  fun x => let x' := #2 GHC.Num.* x in #2 GHC.Num.* x'.

Definition double2 : GHC.Num.Int -> GHC.Num.Int :=
  fun x => #2 GHC.Num.* x.

Definition double1 : GHC.Num.Int -> GHC.Num.Int :=
  fun x => x GHC.Num.+ x.
```

CHAPTER 9

Related Work

I am hardly the first person to do work on verification, verification of Haskell, or verification by translation. In this chapter, I discuss some of the other work in these fields.

9.1. Extraction

Coq natively includes a feature that is essentially the inverse of `hs-to-coq`: *extraction* (Letouzey, 2002; The Coq Development Team, 2020b, sec. "Program extraction"). Extraction is a feature of the Coq system that allows the programmer to generate OCaml, Haskell, or Scheme source code from a Coq program. This can be viewed as a form of compilation, since (pending work on Certicoq (Anand et al., 2017)) Coq does not otherwise have a compiler. Indeed, when running Coq code *for the purposes of running it* – much as we would run a program written in Haskell, OCaml, or Scheme – extraction is the order of the day.

At the same time, extraction is a direct translation between Coq and another high-level functional language. When targeting Haskell, this means that extraction is a sort of `coq-to-hs`: a way to move from a language suited for proof to a language suited to computation. This dual nature means that extraction from Coq *also* leads to mechanically verified Haskell programs; however, the properties of these programs are also turned around from `hs-to-coq`. Where `hs-to-coq` allows Coq verification of existing Haskell programs, extraction instead allows generating Haskell programs from existing verified Coq programs (or unverified Coq programs, if you have one). Thus, extraction cannot be used to increase the number of verified programs in the world – while it can increase the number of verified Haskell programs, it can only do so if the corresponding Coq program already exists.

Letouzey does discuss readability, and make some effort towards that end – while the output often has extra parentheses and odd line breaks (a situation I am very familiar with from `hs-to-coq`), the result in simple cases often remains very "legible", in the similarity sense. However, because the user will not be verifying the output, the need to maintain strict parity is not the same as we have with `hs-to-coq`. Consequently, Letouzey is willing to make certain changes, such as inlining small functions and turning simple wrapper data types into type synonyms (but see Section 9.1.2). One particular loss of legibility comes with type classes. Even when extracting to Haskell, Coq uses the same representation of type classes as it does in OCaml: an explicitly-passed record of functions. This makes interfacing with existing Haskell programs less pleasant (and is one reason we cannot simply "round-trip", running `hs-to-coq`, extracting the result, and comparing it with the original source). However, this is in many ways unavoidable: it means that Coq does not depend on the details of Haskell's

constraint solver. In particular, Coq does *not* guarantee coherence of type classes, and so it accepts strictly *more* type-class–using programs than Haskell does.

On the other hand, this willingness to adapt representations also improves other kinds of legibility. Consider the `sig` type

```
(* From Coq.Init.Specif *)
Inductive sig (A:Type) (P:A -> Prop) : Type :=
    exist : forall x:A, P x -> sig P.
```

The type `sig A P` is the type of all `A`s such that `P` holds of them, and is implemented as a dependent pair. For example, `sig Z (fun n => n > 0)` is the type of all positive integers. Because proofs are erased at runtime (see Section 9.1.1), this is a simple wrapper around `A` at extraction-time, and is thus elided. This means that extracted code that operates on these dependent pairs – a common way to do verification in Coq! – contains no verification cruft at all. Much cleaner!

9.1.1. Making Coq extraction-friendly. Just like `hs-to-coq`, Coq's extraction has to deal with the fact that Haskell (and OCaml, and Scheme) are all thoroughly different from Coq. One principle difference stems from the fact that the target languages' type systems are *weaker* than Coq's (or in Scheme's case, nonexistent). When necessary, Coq is happy to insert unsafe conversion functions to convince OCaml or Haskell that a piece of code is well-typed (`Obj.magic` or `unsafeCoerce`, respectively.) However, there are often simpler things that can be done. For example, in Coq, type arguments to polymorphic functions are exactly like regular arguments; in OCaml and Haskell, polymorphism is strictly at the type level, and in Scheme, it doesn't exist thanks to the lack of types. Extraction therefore must translate polymorphism at the type level only, and elide the arguments at the value level. Similarly, the three sorts – `Set`, `Type`, and `Prop` – are also elided. This sort of elision is essential to compatibility with the weaker type systems of the target language.

One of the biggest purposes of elision during extraction is to completely eliminate the "logical" portions of Coq: those parts that live in `Prop`. By design, Coq splits its universes of proofs and computations apart; proofs live in `Prop` and computations live in `Set` and `Type`. In its full detail, the story is not quite so simple: `Prop` also lives in `Type`, and thanks to dependent types, there's nothing stopping you from writing proof-carrying code in `Type`. But this distinction is baked in to both the culture and the semantics of Coq. For example, both the principle of proof irrelevance and the stronger principle of propositional extensionality are consistent with Coq; the former says that all proofs of a given proposition are the same, and the later says that all truth-conditionally equivalent propositions are the same.

```
(* From Coq.Logic.ProofIrrelevance *)
Axiom proof_irrelevance : forall (P:Prop) (p1 p2:P), p1 = p2.

(* From Coq.Logic.PropExtensionality *)
Axiom propositional_extensionality :
  forall (P Q : Prop), (P <-> Q) -> P = Q.
```

Yet we know these are not consistent for the more computationally minded types that we are used to dealing with: we know that `nat` has more than one inhabitant; and we know that, even though we can construct functions from `bool` to `unit` and vice versa, those two types are not equal. So `Prop` really does have logical properties that the usual computational universe lacks.

Coq takes advantage of this split to omit all "logical" values – anything whose type is in `Prop` will be omitted from the translation if at all possible. For example, consider the following program:

```
Fixpoint div (p q : nat) (nonzero : q <> 0) : nat :=
  (* implementation elided *).

From Coq Require Import Extraction.
Extraction Language Haskell.
Recursive Extraction div.
```

This program defines a division function `div`, and then extracts it: it imports the module allowing for extraction, sets the output language to Haskell, and then extracts the `div` function and all its dependencies. This division function takes two natural numbers, as well as a proof that the denominator is nonzero. This proof lives in `Prop`, and so we know that computation cannot depend on it; consequently, extraction will produce a two-argument function:

```
module Main where

import qualified Prelude

data Nat =
   O
 | S Nat

div :: Nat -> Nat -> Nat
div p q =
  {- the translation of the implementation -}
```

There are two exceptions to the usual rules of elision. First, if a term that would be elided (e.g., a term in `Prop`) is extracted directly, then its type will be extracted to some appropriate trivial type, and the value will be exceptional. In OCaml, the type will be extracted to `Obj.t`, the type of OCaml's internal representation of all objects; in Haskell, `()`, the unit type; and in Scheme, there are no types. In OCaml, the value is constructed to type-check but be unusable; in Haskell, it is simply a call to `error`; and in Scheme, it is a one-argument function that returns itself.

Second, given a function with only elided parameters but a non-elided result (e.g., a possibly-polymorphic function with only logical arguments and a computational result), extraction needs to ensure that the result is not evaluated early. For example,

consider `False_rec : forall (P : Set), False -> P`, which eliminates inconsistent hypotheses in computational contexts. If the `False` argument were deleted, then there would be no arguments left (recall that type arguments are always elided), and the extracted term would have the OCaml type `'P` and the Haskell type `p` (or, more explicitly, `forall p. p`). This is fine in Haskell, but impossible in OCaml! Since OCaml and Scheme are both eager languages, all functions are extracted to have at least one argument, and a dummy argument of the appropriate dummy type is inserted if necessary. Thus, when extracting `False_rec` to OCaml, we get output that expects one argument:

```
type __ = Obj.t

(** val false_rec : __ -> 'a1 **)

let false_rec _ =
  assert false (* absurd case *)
```

However, when extracting it to Haskell, we get output that expects none:

```
module Main where

import qualified Prelude

false_rect :: a1
false_rect =
  Prelude.error "absurd case"

false_rec :: a1
false_rec =
  false_rect
```

(For some reason, extracting to Haskell results in less aggressive inlining – the Coq implementation of `False_rec` is indeed in terms of `False_rect`.)

9.1.2. Customizing extraction. Extraction is both more and less powerful than `hs-to-coq`. More powerful, in that it is guaranteed to work on *any* Coq program; less powerful, in that it is not as customizable. But there are still knobs to tune, and they are similarly important to getting high-quality extracted output. The manual (The Coq Development Team, 2020b, sec. "Program extraction") discusses the various commands that configure the behavior of extraction. There are too many to discuss them all in detail here, but it is a worthwhile comparison to see how these compare to `hs-to-coq`'s edits.

The first category of customizations consists of global changes: setting the extraction language, turning optimizations on or off, and toggling whether or not to inline functions and wrapper types. These are global configuration options; unlike with edits, you cannot specify them for individual definition, only for whole bouts of extraction.

The second category consists of per-definition changes, much like for edits. The three per-definition changes that can be made are:

(1) Toggling inlining for specific definitions, as opposed to the global inlining behavior (not available for wrapper types).
(2) Declaring that extra arguments should be elided, in addition to the type arguments and logical arguments.
(3) Defining replacement extraction output for definitions or types.

The first is the most straightforward; it allows, for instance, inlining `id` and not inlining `andb` no matter what the global settings are. The second and third, however, are a bit more interesting.

Being able to omit extra arguments is done through the **`Extraction Implicit`** directive, and it allows the programmer to declare that some number of named (or numbered) arguments should be omitted. This could be useful, for example, when working in the SSReflect style. SSReflect prefers to encode everything possible in the computational universe, such as by using predicates that compute booleans. One way that the user may do this is by defining predicates in `Type`, instead of `Prop` – there's nothing about `Type` that prevents a proof-like definition. These can be useful within Coq, allowing different forms of computation, but then extraction will refer to them even when unnecessary. Thus, **`Extraction Implicit`** allows removing them. It will correctly halt with an error if an omitted argument is actually used; however, this behavior can be disabled globally for ease of debugging.

Finally, declaring replacement extraction output is most like the **`rename`** or **`redefine`** edit. The **`Extract Constant`** and **`Extract Inductive`** commands allow for the replacement of the autogenerated translation with an alternative. This is particularly important for axioms, which normally are translated, with a warning, to an output-language error – since they lack computational content, they can't be meaningfully extracted.[45] **`Extract Constant`** is almost just that simple – it replaces one name with another, like **`rename`**. However, type constructors must be fully applied, and their arguments must be fully specified. For example, here is how we would take an axiomatized type for I/O effects and an axiomatized print function, and translate them to Haskell's `IO` and `print`:

```
Axiom io : Type -> Type.
Axiom print_nat : nat -> io unit.

Extract Constant io "a" => "IO a".
Extract Constant print_nat => "print".
```

(The eagle-eyed observer will note that the use of `a` as the variable means this is not output-language agnostic. Indeed, we would have to write `"'a"` and flip the type application around to extract to OCaml. In general, when specifying output code as with these commands, the details will be language-specific.)

[45] Logical axioms, as they aren't intended to have computational content, are less of an issue here.

The `Extract Inductive` command is slightly more complex. In the simplest case, it just tells Coq to translate a Coq inductive type into a target-language inductive type. For example,

```
Extract Inductive bool => "Bool" ["True" "False"].
```

will extract Coq `bools` as Haskell `Bools` – the exact opposite of what we do in `hs-to-coq`. The first string after the arrow is the type name, and the strings within square brackets are the constructor names. But what if we don't want the situation to be that simple? In that case, we can specify a third argument after the square brackets, which is an eliminator function to replace pattern matching. This, for instance, allows us to replace an Coq inductive type with a target-language non-inductive type; the example from the manual is to replace Coq `nats` with OCaml integers. The equivalent code in Haskell would be

```
Extract Inductive nat => Int ["0" "(+1)"]
  "(\f0 fS n -> case n of { 0 -> f0 () ; _ -> fS (n-1) })".
```

We can see here that the eliminator expects one function per constructor, and a final argument which is the term being matched on. Each constructor-matching function expects one argument for every argument to the constructor; in the zero-argument case, the function gets an extra argument of type unit, which is not necessary in Haskell but is in eager languages.

9.2. Translating Haskell to non-Coq languages

In contrast to `hs-to-coq`, there are also several "`hs-to-noq`"s out there – converters from Haskell to languages for verification that are not Coq. In particular, there are multiple tools targeting both (1) Isabelle/HOL and (2) the Agda family of languages .

9.2.1. Isabelle/HOL. Isabelle is a computer-assisted theorem prover. Unlike Coq, it is not based on dependent type theory, but instead on logic; in fact, it is *parameterized* over the logic it uses. Most developments are done in Isabelle/HOL, which uses higher-order logic, but there are alternatives. There have been at least two projects that involved translating Haskell into Isabelle/HOL: the Haskabelle project and seL4.

9.2.1.1. *Haskabelle.* Haskabelle (Haftmann, 2010) was a direct parallel to `hs-to-coq` that targeted Isabelle/HOL, as well as being a direct parallel to extraction that goes from Isabelle/HOL to Haskell. Unfortunately, it is no longer maintained, and has since been removed from the Isabelle distribution (see Section 9.2.1.2). Here, I will focus on the translation from Haskell into Isabelle/HOL.

Though Isabelle and Haskell are in many ways less similar than Haskell and Coq – Isabelle is not *just* a functional programming language, the way that Coq is – they are certainly similar *enough*. Haskabelle, like `hs-to-coq`, focused on the question of legibility. During translation, the goal is to have a one-to-one correspondence between Haskell declarations and Isabelle/HOL declarations (when translating in this direction). Isabelle also features type classes, and Isabelle's `eq` type class is automatically identified with Haskell's `Eq` type class.

Unlike `hs-to-coq`, however, Haskabelle does not attempt to cover the entire Haskell language. They target features of Haskell which line up with features of Isabelle/HOL, thus preserving legibility at the cost of comprehensively covering Haskell. The two features they call out as not being supported are guards (which we also had trouble with) and "arbitrary bindings", which refers to things like partial pattern matches in `let` expressions.

Haskabelle also features similar, although less comprehensive, abilities to control the translation of individual definitions. For example, when translating from Isabelle/HOL to Haskell, it is possible to completely and safely change the definition of a function: by annotating a lemma with the `code` attribute, the user can provide an alternative set of equations defining a function, which will then be translated to Haskell. Because this comes from a proven lemma, we know this will not change the meaning of the translated code.

Haskabelle doesn't have the same emphasis on never editing code; for example, their analog of our `termination fn` edit is to add a `{-# HASKABELLE permissive fn #-}` pragma above the definition of `fn`. This produces Isabelle/HOL output containing the keyword `sorry` where a termination argument needs to go.

Much like `hs-to-coq`, Haskabelle had to make concessions to its target language during translation. For example, local function definitions in Haskell were automatically refactored to top-level definitions in Isabelle/HOL. Sometimes, Haskabelle performed these operations, which they call *adaptations*, for simpler alignment; for example, connecting the built-in list type and functions between the two languages.

Sometimes, the features of Isabelle/HOL are actually *more* useful than Coq's for this translation. One key example of this we have already seen back in Section 3.8: while Isabelle is also a total language, it also, in the logical as opposed to type-theoretic tradition, has only *inhabited* types. It is also *classical*: every type in Isabelle not only has at least one inhabitant, but we can pick out an undifferentiated inhabitant of any type with the special constant `undefined`. Because in Isabelle this is *consistent*, Haskabelle can use this to translate partial patterns without any problems.

Continuing in the vein of totality, Isabelle also has more automation around certain features, including termination. Isabelle's `fun` keyword defines a function and attempts to automatically find a termination proof for it; Haftmann reports that "[t]he automation of `fun` succeeds in lots of cases", which `hs-to-coq` can only envy.

Haskabelle also chose to validate their work on a library implementing finite maps (Adams, 1993); the library is not `containers`, but it too implements finite maps in terms of balanced binary trees. This library, `FiniteMap` (Iborra, 2007), is much simpler than `containers` (and has since been deprecated in favor of it); it consists of a single literate Haskell file with 294 nonblank noncomment lines of code. Consequently, Haftmann was able to verify the entire thing. Rather than an edit file, he had to make just five substantial changes: remove all literate comments; inline the definition of `isJust` from another module; replace guards with `if` statements; replace a partial pattern match in a `let`-binding with a `case` expression; and dropping the `Eq` instance. As we have seen from our need for substantial edit files, this is an excellent performance!

9.2.1.2. *When was Haskabelle present?* There was no simple announcement of Haskabelle being discontinued, but we can check the historical list of Isabelle releases.[46] This has all releases from 2008–2019 (the current 2020 release is on the home page). We can see signs of Haskabelle from 2008–2016, before it seems to have been removed in the 2017 release of Isabelle. From the 2008 release through the 2016-1 release, the site has a Haskabelle page, but not in 2017.[47] We can also check what we would get by downloading Isabelle: in 2008, there is no sign of Haskabelle. The 2009[48] and 2009-1[49] releases contain a reference to Haskabelle on the downloads page. From the 2009-2 release through the 2016 release, we can also see Haskabelle when we download Isabelle from the downloads page (2009-2 through 2011-1[50]) or the installation page (2012 through 2016[51]). It is in the `contrib/` directory of the top listed tarball, and then ceases to be in the 2016-1 release[52] and beyond.

9.2.2. seL4. The seL4 microkernel is a fully formally verified microkernel that uses capabilities for security and access control (Derrin, Elphinstone, Klein, Cock, and Chakravarty, 2006; Elphinstone, Klein, and Kolanski, 2006; Elphinstone, Klein, Derrin, Roscoe, and Heiser, 2007; Heiser, Elphinstone, Kuz, Klein, and Petters, 2007; Klein, 2009; Klein, Andronick, Elphinstone, Heiser, Cock, Derrin, Elkaduwe, Engelhardt, Kolanski, Norrish, Sewell, Tuch, and Winwood, 2010). While this might sound a bit far afield from `hs-to-coq`, there is in fact a very direct connection: the verification includes an Isabelle/HOL specification derived from executable Haskell code. The seL4 microkernel development started with a Haskell prototype (or implementation) of the desired kernel behavior. Then, through manual refinement, an equivalent C implementation was written and verified against two higher-level specifications. The highest level of specification talks about functional correctness, and is written in Isabelle/HOL. The intermediate level is an executable Isabelle/HOL specification that corresponds to the Haskell prototype, and was in fact generated semi-automatically therefrom.

This all means that seL4 contains three hand-written artifacts: the Haskell implementation of the kernel (the prototype); the C implementation of the kernel; and the Isabelle/HOL abstract specification (which, although generated, is then manually modified). For our purposes, we are focused on only the first: how does the seL4 team work with this Haskell code, and how do they bring it into Isabelle/HOL?

---

[46] At http://isabelle.in.tum.de/download_past.html.

[47] At https://isabelle.in.tum.de/website-Isabelle2008/haskabelle.html and https://isabelle.in.tum.de/website-Isabelle2016-1/haskabelle.html, as well as all the intervening years, but not afterwards.

[48] https://isabelle.in.tum.de/website-Isabelle2009/download_x86-linux.html

[49] https://isabelle.in.tum.de/website-Isabelle2009-1/download_x86-linux.html

[50] https://isabelle.in.tum.de/website-Isabelle2009-2/download.html, https://isabelle.in.tum.de/website-Isabelle2011/download.html, https://isabelle.in.tum.de/website-Isabelle2011-1/download.html

[51] At https://isabelle.in.tum.de/website-Isabelle2012/installation.html through https://isabelle.in.tum.de/website-Isabelle2016/installation.html, for every intervening year as well.

[52] https://isabelle.in.tum.de/website-Isabelle2016-1/installation.html

Their translation of Haskell into Isabelle/HOL echoes many of the same themes we saw in `hs-to-coq`, leveraging both conceptual and syntactic similarity. They care about "legibility" in the sense of Haskell-Isabelle/HOL similarity, and they care even more than us about making sure the generated output is comfortable to work with as a specification. On the other hand, unlike `hs-to-coq`, the authors discuss the translation much more in terms of textual transformations, as opposed to operations on abstract syntax trees. They call out regular expressions as a component of the translation; discuss token-by-token transformations (e.g., turning Haskell's `->` into Isabelle/HOL's `=>`); and even say that "[f]or many basic terms no translation is required". This means that they can and do aim, not just for structurally similar Isabelle/HOL and Haskell code, but *textually* similar, in terms of the layout of the generated code. They also are not interested in guaranteeing that the output of their translation is one-and-done; instead, they are ready and willing to manually adjust the generated Isabelle/HOL as part of their development process. Even so, they at one point had approximately 90% of the translation work automated (Elphinstone et al., 2006) – another similarity with `hs-to-coq` is that they are happy to have "good enough" techniques, although their happiness is with translation and ours is with verification.

This all connects to the fact that the seL4 translation from Haskell to Isabelle/HOL is bespoke and closely tied to the particular Haskell they are translating. They have no expectation that their translation handle arbitrary Haskell code, and no need for it to, either. This reduces some of the burden of dealing with mismatches between Haskell features and Isabelle/HOL features. Some features do line up well regardless, such as partiality or `undefined` corresponding to Isabelle/HOL's `undefined` as we saw above. Sometimes, seL4 simply does not use those features; for example, Isabelle/HOL does not have recursive `let`-bindings, and its list comprehensions are weaker than Haskell's. These would be a problem for GHC, but they simply do not occur in the seL4 kernel prototype. Other features need to be syntactically adjusted, as we have seen with, e.g., guards or record selectors in `hs-to-coq`; some examples for seL4 are nested patterns or, for them as well, record selectors. To handle complex patterns, the seL4 translation turned them into predicate tests followed by extracting the bound variables. Their record selector desugaring is similar to `hs-to-coq`'s.

Nevertheless, as is inevitable, there are still differences that cause trouble. Three features they highlighted as problematic were nontermination, monads, and the `Dynamic` type. Nontermination is the same issue `hs-to-coq` has, the same issue Haskabelle had, and indeed just a general issue when doing any sort of verification. For the most part, Isabelle/HOL was able to handle all the termination proofs automatically; in the one case where they had a challenging termination argument, they were able to provide a slower-than-necessary one directly. This demonstrates that Isabelle/HOL's termination checker is evidently much more powerful than Coq's, as the termination argument was "machine words are finite, and this algorithm will visit each pointer in the data structure once" – would that we could get Coq to accept such an argument!

Monads were a minor issue because Isabelle/HOL supports *type* classes, but not *constructor* classes (in fact, it does not support abstraction over type constructors at all (Huffman, Matthews, and White, 2005)). This is a distinction most in the Haskell

world have simply forgotten about, but the difference is that type classes are parameterized over types, and constructor classes are parameterized over type constructors (or more complex things); a type class in Haskell has kind `Type -> Constraint`, whereas a constructor class has kind `(Type -> Type) -> Constraint`, or more generally kind `K -> Constraint` for any `K` more complex than `Type`. The `Monad` type class is naturally a constructor class, and so Isabelle/HOL cannot talk about monads in general. This is unfortunate for seL4, as the kernel is implemented in terms of monad transformers: all operations are in the `State` monad transforming the kernel state, and some have error monad transformers layered on top. The solution here was simply to work monomorphically for monads; with only three cases (`State` plus 0, 1, or 2 error monads), the duplication was trivial, and there was minimal proof duplication necessary. Indeed, the seL4 developers reported being happier with the clarity that monomorphization provided, not despite but *because* it meant distinguishing `do` in the state monad from `doE` in the error-over-state monad.

Finally, the `Dynamic` type in Haskell can be safely created from or cast to a value of an arbitrary type, by providing a `Typeable` constraint. As we've seen in `hs-to-coq`, providing for these complex Haskell type system features in languages that don't support them can be tricky, and sometimes you need to provide your own implementations of missing libraries. The seL4 developers did just that, and provided a version of `Dynamic` in terms of encoding the value into a list of bytes. This functionality covered all the types and behavior they needed, at the small cost of having to write some of it themselves.

One interesting decision point that the seL4 developers made was to use Isabelle/HOL instead of Isabelle/HOLCF. Isabelle is, as mentioned above, parameterizable over the underlying logic, and the choice was between straightforward higher-order logic (HOL) and a logic of computable partial functions (HOLCF). They chose HOL and not HOLCF for the same reasons `hs-to-coq` chose recursion and direct use of values over pervasive corecursion or an `ERR` monad (see Section 3.8) – they work with total code, and they don't want extra hassle. If you want to prove that all of your functions are total *anyway* – and they did – then allowing partial functions isn't beneficial; and having to reason about continuity and bottom values and such just makes your life harder for little to no gain. While they considered (Elphinstone et al., 2006) using Programatica (Hallgren, Hook, Jones, and Kieburtz, 2004), which targeted HOLCF, to automate the translation, it was not able to translate their code base. (For more on Programatica, see Section 9.2.3.1.)

One very interesting difference between how seL4 uses translated Haskell and how `hs-to-coq` works with it is that they conceive of the Haskell as a *specification* for the actual artifact, which is the C microkernel. The Isabelle/HOL generated from the Haskell code is the target of a refinement proof with respect to the C code. Except that at the same time, this generated Isabelle/HOL sits *below* the abstract functional-correctness specification, and is itself refined against it! So the (translated) Haskell code is both verifier and verified, a very different model than `hs-to-coq` works with.

9.2.3. Agda, Alfa, and Programatica. Agda is a dependently-typed programming language and proof assistant. Unlike Coq, it has a more direct focus on writing dependently-typed *programs*; nevertheless, it too is total and thus suitable for theorem proving. It has gone through multiple major versions, and is descended from an earlier language, Alfa. For many of the same reasons that `hs-to-coq` chose Coq as a target language, there has been much work over the years focused on verifying Haskell using Agda (or Alfa) as a target language. One locus of this work was the Programatica project (Hallgren et al., 2004); there has also been work outside of it.

9.2.3.1. *Programatica.* The Programatica project (Hallgren et al., 2004; The Programatica Team, 2003) was a large-scale project focused on writing trustworthy, validated software. This effort resulted in a large software toolset focused on validating Haskell code. The scale of their efforts was large; they were interested in building "a framework for *Extreme Formal Methods*" (Hallgren et al., 2004) by analogy with extreme programming. The goal was to enable validation-program codesign, ensuring that formal methods were present from day 0 of a development project.

The tooling support was not focused solely on formal *verification*. The idea was that Haskell code could be augmented with property and assertion declarations at the top-level, where a function definition could go (Kieburtz, 2002). The Programatica development environment could then annotate the assertions with *certificates*, describing why these assertions could be trusted. These certificates could be backed by different kinds of evidence:

(1) "I say so" evidence, their term for evidence that is simply a bare statement that the assertion is true.
(2) Individual test cases that validate (i.e., provide evidence for) the assertion.
(3) Passing QuickCheck tests.
(4) A formal proof in Alfa.
(5) An automatic proof in P-logic, a program logic for Haskell, using a verifier called Plover.

As we can see from this selection of evidence types, Programatica was about more than just total formal verification; it was about the broader question of building formally *validated* software using any methods that could increase confidence. We will not further discuss forms of evidence 1–3, as they are relatively straightforward. However, using Alfa and Plover are both more involved, and using Alfa in particular is directly germane to `hs-to-coq`.

Verification in Alfa. In order to write a proof about a piece of Haskell code in Alfa, of course, one needs to *translate* that code to Alfa somehow, and the Programatica project took the same approach as `hs-to-coq`: translate Haskell source code to Alfa source code, and then use all the typical features of Alfa to verify the result. Just as with `hs-to-coq`, their translation has to deal with mismatches large and small between Haskell and Alfa, many of which are reminiscent of the issues we had to address, such as moving from two namespaces (types and terms) to one. One unusual difference is the way they had to deal with termination: while Alfa has a termination checker, they are still able to translate non-structurally recursive functions and simply obtain a partial correctness result. Like `hs-to-coq`, they discuss needing to remove

syntactic features such as list comprehensions (unlike `hs-to-coq`, this includes type classes), as well as the similarity between the input and the output despite this.

`hs2alfa`. On Thomas Hallgren's website,[53] there is also reference to a tool called `hs2alfa`, which may be the same translation tool discussed above. The web page makes very clear how many of the challenges we faced in developing `hs-to-coq` are not unique, but genuine pain points in the translating-Haskell-to-a-theorem-prover space. The "Limitations" section includes several limitations that are almost identical to issues we have seen in our exploration of `hs-to-coq`. Some are features it does not support that we support but did have trouble with: guards that use pattern-matching; record field selectors; Haskell names that are keywords in the target language. One is smaller, but familiar: he mentions the issue of name clashes when combining mutually-recursive modules into a single module.

Lastly, and to me most strikingly, they too encountered the issue of type variable name collisions when translating type class instances. The `hs2alfa` website says, under "Other problems":

> Name capture can occur, for example, in the translation of
>
> ```
> class Functor f where fmap :: (a->b)->f a->f b
> instance Ix a => Ix (Array a) where fmap = ...
> ```
>
> there will be a problem with capture of the type variable a.
>
> —Thomas Hallgren

(Sic: the typos of `Ix` instead of `Functor` on the second line are in the original.) Years later, without having read this, I documented the internal behavior of the function `HsToCoq.ConvertHaskell.Declarations.Instances.makeInstanceMethodTy` with the following comment:[54]

> GOAL: Consider
>
> ```
> class Functor f where
>   fmap :: (a -> b) -> f a -> f b
> instance Functor (Either a) where fmap = ...
> ```
>
> When desugared naïvely into Coq, this will result in a term with type
>
> ```
> forall {a₁}, forall {a₂ b},
>   (a₂ -> b) -> f (Either a₁ a₂) -> f (Either a₁ b)
> ```
>
> Except without the subscripts! So we have to rename either the per-instance variables (here, `a₁`) or the type class method variables (here, `a₂` and `b`). We pick the per-instance variables, and rename `a₁` to `inst_a₁`.
>
> ASSUMPTION: type variables don't show up in terms. Broken by ScopedTypeVariables.
>
> —Antal Spector-Zabusky, the module `HsToCoq.ConvertHaskell.Declarations.Instances`, lines 377–394

It is remarkable how similar these two issues are, down to the choice of example!

[53] http://ogi.altocumulus.org/~hallgren/Programatica/tools/hs2alfa/

[54] I have replaced "ALL CAPS" with SMALL CAPS for legibility.

P-*logic and Plover.* The final form of evidence that Programatica considers is automatic proofs in *P*-logic. *P*-logic is an alternative approach to verifying Haskell by providing a program logic for it (Harrison and Kieburtz, 2005; Kieburtz, 2002). *P*-logic is a program logic specifically for Haskell 98; it takes particular care in formalizing the notions of strict vs. nonstrict evaluation, particularly in pattern matching and as regards `seq`. Harrison and Kieburtz establish both a model for the logic and its soundness with respect thereto. *P*-logic can be used to prove properties of partial Haskell 98 programs, not just total ones. In the context of Programatica's evidence (Hallgren et al., 2004), they provide the tool Plover to automatically attempt to prove the *P*-logic assertions that have been embedded in the Haskell code. Like most automated theorem provers, Plover is intended to be sound but not complete; it attempts to prove as much as it can about the target Haskell programs, but the presence of other forms of evidence means that there are alternatives if it cannot automatically find a proof. Plover (as opposed to *P*-logic itself) does not cover all of Haskell 98; it cannot reason about recursion or data type declarations, among other things. Nevertheless, this sort of automated proof is a powerful alternative strategy for verifying properties of programs in general.

9.2.3.2. *Translating Haskell into Agda/Alfa.* Other work besides the Programatica project has enabled verifying Haskell through translation into Agda-family languages. Dybjer, Haiyan, and Takeyama presented a verification methodology that combines multiple different forms of validation: property-based random testing, formal mechanical verification by translation into Alfa/Agda, and model checking (Dybjer et al., 2004). The broad idea is to apply random testing to guide the development of proofs, and use proofs to guide the application of tests. For example, a top-level function may have a specification that random testing refutes, but this does not expose where in the *implementation* the problem occurred, particularly if the top-level function is implemented in terms of local functions. In this case, proofs may guide the user to see the split-up specifications for these local functions, which can then be tested and produce more informative counterexamples on failure. In the other direction, getting a counterexample from a random test is a *much* more efficient way to discover that a specification is wrong when compared to trying and failing to write a proof, so ensuring active use of random testing during the verification process is incredibly helpful.

The random testing is further enhanced by model checking. Dybjer et al. provide a model checker to determine whether a boolean formula is a tautology, and then actually use this as the target of random testing. Rather than properties resulting in a boolean, they instead result in a query, "is this boolean expression a tautology?". During the random testing loop, after plugging in the generated parameters, whether the test passes is determined by whether or not the resulting boolean expression – which may refer to the generated variables – is in fact a tautology.

Just as with every other translation we have seen, Dybjer et al. need to address the question of partiality. They take a different tack than any other translation we have thus far examined, including `hs-to-coq`: they add an extra proof argument to every partial function that specifies when it can be called. For example, the Haskell function `fromJust`, defined by

```
fromJust :: Maybe a -> a
fromJust (Just x) = x
fromJust Nothing  = error "fromJust: Nothing"
```

is only safe to call on a value of the shape `Just x`, which is checked by the predicate `isJust`. Consequently, the translation to Alfa/Agda would be the equivalent of the idealized dependent Haskell definition

```
fromJust :: (mx :: Maybe a) -> (pf :: T (isJust mx)) -> a
fromJust (Just x) = x
fromJust Nothing  = case pf of {}
```

Here, `T` is a type that is inhabited if and only if its argument is `True`:

```
data T :: Bool -> Type where
  TT :: T 'True
```

The translation does not automatically determine these predicates, unsurprisingly; they must be specified by the user, and then they are propagated throughout the call chain. Functions may have any number of these proof parameters added, but otherwise they are not modified.

This approach is in fact dogfooded – the first example they choose to verify in Alfa/Agda is the model checker that is a part of the system. They then continue by applying this methodology to a bitonic sorting algorithm. Although this algorithm is polymorphic, they take advantage of parametricity (proving the appropriate theorem in Alfa/Agda) to reduce the proof obligation to sorting only lists of booleans. Because they are then operating over booleans, they can then verify correctness with their model checker.

9.2.3.3. *Translating Haskell into Agda.* Finally, we come to a project that involves a language simply named Agda, albeit a version that is still very different from modern Agda. Abel et al. present a technique for verifying Haskell programs in Agda through translation, but this time from *Core* and not from Haskell (Abel et al., 2005). This renders a great many problems moot at one stroke: Core is much smaller than Haskell, so the number of features they need to support is vastly reduced, and leaves out type classes entirely; everything is explicitly-typed; there is no question about whether or not they can support all of Haskell; and the need for strict similarity to the input is reduced. The downside is that strict similarity to the input *is* reduced – the logical reasoning must be done in terms of compiled Core, and not the original Haskell program. Nevertheless, Abel et al. report that in practice, one can still think in terms of the Haskell code, since the Haskell and Core are semantically equivalent; and indeed, we have certainly experienced an analogous phenomenon when working with `hs-to-coq`, where some of us (myself included) think in terms of the original Haskell code rather than the generated Coq code.

Even though Core is much simpler, there are still enough differences with Agda to cause problems. Some are small, such as the Core containing wildcard patterns and Agda requiring an explicit exhaustive list of patterns. A bigger problem is that Agda only permits `case` expressions (1) as the outermost expression in a function definition, and (2) applied to a single variable. It is not permitted to write

`f (case x of ...)`, nor is it permitted to write `case (f x) of ...`. Thus, every (non-trivial) `case` expression has to be translated to a local function: `case e of ...` becomes `let f x = case x of ... in f e`.

Another problem is, of course, partiality. Again, we see that termination is not an issue when it comes to infinite loops – this version of Agda requires termination for soundness, but does not check for it. Thus, Abel et al. simply join in with this situation; absent a separate termination proof, all their results are partial, but they do not let this worry them. The bigger issue is partiality from sources such as `error` or partial pattern matches. Instead of trying to preserve the partiality with minimal changes, they instead make the most invasive possible change, and *make the entire translation monadic*. Function application is replaced with bind everywhere, literals are wrapped in `return`, and every type is wrapped in `m`. This translation is then optimized slightly further, to avoid too much noise – top-level functions are given simpler types of the form `m a -> m b -> m c`, rather than `m (m a -> m (m b -> m c))`. The translator then handles adding extra monadic layers if the function is ever used in a context that expects such a thing.

Another complication, which we have not yet seen, is *impredicativity*. For this translation, it is actually an issue that Haskell validates the inconsistent axiom `Type :: Type`! This is an issue in a few ways. First, the way that Core encodes GADTs can lead to instantiating a type variable with a polymorphic type, which is not permitted in Agda without raising the universe level. This is not a problem Abel et al. solve, as they find it sufficiently uncommon. However, two problems remain that they do solve. The first is that type class dictionaries are actually quite similar to GADTs in this way; for example, a type class like `Functor` is represented, at the Core level, as a dictionary with the polymorphic field `fmap :: forall a b. (a -> b) -> f a -> f b`. Second, because polymorphic types are one universe level higher than the types they quantify over, the monadic translation would cause impredicativity issues if it were ever applied to a polymorphic type. If we were translating a Haskell function of type `forall a. P a -> Q a`, then naïvely we would end up with the type `m ((a :: Set) -> m (m (P a) -> m (Q a)))`, since `forall` is just a type lambda. But now we see that `m` is applied to types in `Set`, like `P a`, as well as types in the next universe up, like `Set` itself.

To resolve these issues of impredicativity, the translation into Core distinguishes between arrows that *can* fail and arrows that *cannot* fail. Given the semantics of Haskell, it is always safe to apply a function to a type variable; this will never cause a crash, and so we can eliminate the monadic wrapper after `forall`s. To resolve the further issue with type class dictionaries, they observe that the polymorphic type variables are only ever instantiated with monomorphic types, resolving the impredicativity.

One important feature of the monadic translation is that it has been left *abstract*. Rather than translating into some particular error monad, Abel et al. explicitly left the translation polymorphic in the monad. This way, one can either work in the identity monad, if working with total code; or the maybe monad, if working with partial code. The identity monad they use is the true identity monad without a newtype wrapper, and so if `m` is instantiated like that, the monadic noise really disappears. On the other

hand, values like `undefined` must be translated to `Nothing`, and so sometimes the specific monad is determined. This monadic structure gives this translation the most flexibility we have yet seen when it comes to reasoning about partial code.

9.3. Translating functional languages into logical formulæ

A plain shallow embedding is not the only possible way to embed a functional programming language into a theorem prover. While deep embeddings are of course always possible, another powerful technique is to translate functional (or other) programs into *logical formulæ* in the theorem prover, perhaps in terms of a domain-specific language, and then reason about these logical formulæ. Though this technique does not produce as close a correspondence to the original input as the shallow embedding techniques we have been discussing, theorem provers are very good at reasoning about logical formulæ.

Thompson describes a translation from the lazy functional language Miranda to logical formulæ (1995). This translation is presented on paper, as well as formalized within Isabelle. The formulæ produced specify the particular values functions evaluate to, equating function applications to their right-hand sides. This naturally necessitates the statements being phrased as conditionals, in order to encapsulate the effects of pattern matching and guards (a particularly thorny part of the translation). In order to capture both the partiality and the laziness of Miranda, this translation works over bounded partial orders; there is a bottom element in every translated type, allowing the user both to prove theorems about partially-defined values and to prove theorems about partial functions.

Thompson also produced a very similar translation for Haskell (1992). This translation is broadly similar, but does run into some unique-to-Haskell complexities. The pattern and expression language is richer: for example, irrefutable patterns (such as `~(Just x)`) can cause expressions far away from the pattern to evaluate to $\bot$; and `case` expressions require axiom schemes to evaluate, rather than all pattern matching being relegated to functions and thus eliminated by the transformation, as in Miranda. Type classes also present a challenge, and are handled by introducing "logical classes", logic-level classes containing *theorems* about the Haskell type classes (similar to our `ClassLaws` type classes, which we saw back in Section 5.1.1).

Finally, Charguéraud produced a translation from pure OCaml programs into logical formulæ in Coq (2010). Unlike Thompson's approaches above, the resulting formulæ are not assertions of equality, but *characteristic formulæ*: assertions parameterized by postconditions that totally characterize the behavior of functions. We write $[\![\texttt{t}]\!]$ for the characteristic formula corresponding to a pure OCaml term `t` which when translated has Coq type `T`; this characteristic formula then has type `(T -> Prop) -> Prop`. This type can be thought of as the set of all valid postconditions for `t`; naturally, computing the returned `Prop`osition takes more computation than that might otherwise suggest. Despite the logical formulation, Charguéraud also takes care to ensure that translated programs come out as a legible, similar-looking formula through the use of carefully-designed syntactic sugar. This focus on readable output carries over to the formulation of specifications and even to the tactic library for working with these characteristic

formulæ. Charguéraud was able to apply these techniques to verify more than half of the data structures from Okasaki's *Purely Functional Data Structures* (1999).

9.4. LiquidHaskell: an alternative approach to verifying Haskell programs

While `hs-to-coq` presents one means of verifying existing Haskell programs, it is by no means the only game in town. LiquidHaskell is a verification tool developed by Vazou, Seidel, Jhala, Vytiniotis, and Peyton Jones (2014); Vazou, Tondwalkar, Choudhury, Scott, Newton, Wadler, and Jhala (2018b) which builds on the technique of *liquid types* (Rondon, Kawaguchi, and Jhala, 2008). Liquid types are a form of *refinement types*, allowing the programmer to constrain ordinary type signatures, such as `head :: [a] -> a`, to more precise type signatures, such as `head :: {xs : [a] | len xs > 0} -> a`. This type signature records that the input to `head` must be a nonempty list – i.e., its length must be strictly positive – which is what is required for `head` to be a safe (that is, total) function. LiquidHaskell augments Haskell with liquid types (hence the name), and allows for the SMT-aided automatic verification of the refined Haskell types.[55]

LiquidHaskell is the ongoing project most related to `hs-to-coq`, and has also seen use on real Haskell code bases; while different from `hs-to-coq`, it is very exciting to see another project exploring this space. Due to this similarity, I discuss LiquidHaskell in more detail than other related work, discussing the following in this section.

(1) I begin by contrasting LiquidHaskell and `hs-to-coq`, exploring the relative strengths and weaknesses of the two approaches (Section 9.4.1).
(2) Once we have seen this, I explore the way LiquidHaskell works in more detail (Sections 9.4.2–9.4.4).
(3) Then, now that we know how LiquidHaskell works, I work out a simple example of verifying that lambda calculus terms are well-scoped (Sections 9.4.5 and 9.4.6).
(4) Finally, I discuss how LiquidHaskell has been applied to larger programs, focusing on those applications that connect to work that we have done with `hs-to-coq` (Section 9.4.7).

9.4.1. Comparing and contrasting LiquidHaskell with `hs-to-coq`. As briefly outlined above, LiquidHaskell and `hs-to-coq`, present two very different approaches to the same goal. Instead of the automatic verification provided by SMT solvers that LiquidHaskell has, `hs-to-coq` has manual Ltac proofs. Instead of reasoning about Haskell programs directly, `hs-to-coq` uses edit files to enable reasoning about Haskell programs in a foreign language (Coq). These approaches are both powerful enough to verify real code; at the same time, being so different from each other, they naturally come with different strengths and weaknesses. While not about `hs-to-coq` in particular, Vazou, Lampropoulos, and Polakow (2017) discussed the core of these differences in "A tale of two provers: verifying monoidal string matching in Liquid Haskell and Coq" where they verify a parallel string-matching algorithm both in LiquidHaskell and in Coq. Their paper discusses the tradeoffs from using

[55]This first paragraph, as well as Sections 9.4.2–9.4.6, are originally from my WPE II.

those two languages as provers; while this does not get at the Haskell-specific angle of `hs-to-coq`, it nevertheless provides an important perspective on the differences.

On the quantitative side, Vazou et al. (2017) found that both approaches required similar amounts of specification (285 for LiquidHaskell[56] vs. 248 for Coq); it is harder to compare the size of the executable portions of code, since the Haskell code came with input/output features such as printing, but given that excess the two seem comparable (180 for LiquidHaskell vs. 122 for Coq). However, the amount of proof is a clear win for LiquidHaskell, requiring only 669 lines of proof vs. 766 for Coq, a $\frac{1}{8}$ reduction. This makes sense, as one of LiquidHaskell's advantages is the SMT solver; using powerful techniques such as *proof by logical evaluation* (see Section 9.4.3 for more details) allows it to take advantage of the SMT solver's ability to do proofs for us in a wide domain. Because the Coq code was a port of the Haskell code, it is not useful to compare the time taken to carry out the two different proofs.

On the qualitative side, Vazou et al. (2017) found that LiquidHaskell and Coq were different experiences. LiquidHaskell's use of SMT solvers gives it the ability to complete proofs for you; in my experience, this can feel almost magical. However, because SMT solvers say only "yes" or "no", LiquidHaskell can do no more than point to a failing proof and say "you have an error here". Working in Coq, on the other hand, involves live proof development, and while it is possible to get stuck, you can always see the proof state in order to make decisions about how to move forward. Coq does have some automation available in Ltac, such as the `lia` tactic which solves linear integer arithmetic equations (The Coq Development Team, 2020b, "Micromega: solvers for arithmetic goals over ordered rings"[57]). However, these tactics are more specific and often not as optimized or complete as SMT solvers.

Vazou et al. (2017) also found that writing the code was different. LiquidHaskell favors an intrinsic verification style, with the refinements attaching proofs directly to terms; Coq often favors an extrinsic verification style, and `hs-to-coq` practically demands it with its separation between translated code (program) and hand-written code (extrinsic proof). It is thus no wonder that their Coq development, like ours, required the use of proof irrelevance. And of course, termination checking in Coq was a problem and appeasing it required complex code transformations; in LiquidHaskell, only reflected terms needed to be terminating, and all termination is given by a termination metric, making the situation much simpler.

Finally, one key difference that Vazou et al. (2017) found does *not* apply for `hs-to-coq`: The difference between working in a full-featured general-purpose language (Haskell) and a language designed for verification purposes (Coq). In their comparison, LiquidHaskell had to deal with the lack of a library of utilities for writing proofs, while Coq had to deal with a lack of libraries in the more conventional programming sense (as well as with having efficiency and true parallelism accessible only through extraction). By using `hs-to-coq`, we can get the benefits of both at the same time, by bringing the libraries of general-purpose Haskell code into Coq. This does not come

[56] We are using the numbers for "Liquid Haskell with PLE [Proof by Logical Evaluation]", as we are interested in comparing with the most powerful form of LiquidHaskell; for more on PLE, see Section 9.4.3.

[57] This chapter is available at `https://coq.inria.fr/refman/addendum/micromega.html`.

for free, of course; we must write edit files to get the translation to work. But once we do, this is the key advantage of `hs-to-coq`: the ability to live in both worlds, and to run the original efficient Haskell code having verified it in Coq. One feature they highlight as a negative of working in Coq, however, we still cannot quite avoid: as a general-purpose language, Coq had a much harder time dealing with nonstructural recursion, and `hs-to-coq`'s edits only partially ameliorate that.

9.4.2. An introduction to LiquidHaskell. Now that we have some context for how LiquidHaskell compares with `hs-to-coq`, let's approach it in more detail on its own terms. In general, given a Haskell function with a type of the form

```
func :: A -> B -> C
func x y = ...
```

LiquidHaskell permits augmenting its type to become

```
{-@ func :: {x:A | φ1(x)}
         -> {y:B | φ2(x,y)}
         -> {z:C | φ3(x,y,z)} @-}
func :: A -> B -> C
func x y = ...
```

The refined LiquidHaskell type is given within the special `{-@ ... @-}` comment form, as are all LiquidHaskell directives. Each of the ϕ_i is a Boolean-valued formula with the given free variables. It indicates that the first argument to `head` is not merely an `A`, but an `A` that satisfies the predicate ϕ_1, and similarly the second argument satisfies ϕ_2; the result is guaranteed to satisfy ϕ_3. Because the refinement types come with names, the function arrow here is *dependent*: we can refer to the first argument in the refinement of the second. Should we not need to refine a type, we can always drop the braces, writing just `x:A -> ...`, or even drop the name as well, simply writing `A -> ...`.

However, the refinements ϕ_i are, as indicated by the choice of notation, not simply Haskell terms. Sadly, the halting problem (Turing, 1937) would prevent us from automatically verifying these arbitrary conditions over Turing-complete terms. Instead, the refinements are terms that can be fed to an SMT solver, which can automatically verify or reject them. The grammar of refinements is split into (1) expressions, and (2) Boolean-valued predicates over expressions; the core of the grammar is presented in Figure 9.1 (Jhala, Vazou, Seidel, and other contributors, 2020).[58]

In addition to the integer arithmetic expressions, which SMT solvers can readily deal with, the powerful features we see are variables and function application. What names are in scope, and what functions can be applied? Initially, in LiquidHaskell, we could only use the variables bound by the dependent refinement function arrow, and a special class of functions called *measures*. A measure `m` is a function defined on an inductive data type `T` with constructors `C_1` through `C_N` such that `m`:

[58]The grammar includes additional productions, such as lambda expressions, but these appear to be infrequently used; none of them transform the expressive power of the language. (For example, lambda expressions do not show up in the predicate fragment of the grammar.)

$$
\begin{array}{rcl}
\textit{expr},\ e & ::= & x \\
& \mid & \text{NUMBER} \\
& \mid & \texttt{true} \\
& \mid & \texttt{false} \\
& \mid & \texttt{if}\ p\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & \texttt{-}e \\
& \mid & e\ o\ e \\
& \mid & e\ e \\
& \mid & \texttt{(}\ e\ \texttt{)} \\
\textit{pred},\ p,\ \phi & ::= & \texttt{true} \\
& \mid & \texttt{false} \\
& \mid & \texttt{if}\ p\ \texttt{then}\ p\ \texttt{else}\ p \\
& \mid & e\ r\ e \\
& \mid & e\ e \\
& \mid & \texttt{\textasciitilde}\ p \mid \texttt{not}\ p \\
& \mid & p\ b\ p \\
& \mid & \texttt{(}\ p\ \texttt{)}
\end{array}
\qquad
\begin{array}{rcl}
x & ::= & \text{VARIABLE} \\
\textit{op},\ o & ::= & \texttt{+} \\
& \mid & \texttt{-} \\
& \mid & \texttt{*} \\
& \mid & \texttt{/} \\
& \mid & \texttt{mod} \\
\textit{rel},\ r & ::= & \texttt{==} \mid \texttt{=} \\
& \mid & \texttt{/=} \mid \texttt{!=} \mid \\
& \mid & \texttt{<} \\
& \mid & \texttt{<=} \\
& \mid & \texttt{>} \\
& \mid & \texttt{>=} \\
\textit{bop},\ b & ::= & \texttt{\&\&} \\
& \mid & \texttt{||} \\
& \mid & \texttt{=>} \mid \texttt{==>} \\
& \mid & \texttt{<=>}
\end{array}
$$

FIGURE 9.1. Important parts of the grammar of Liquid Haskell refinements. A refinement is a predicate. (Jhala et al., 2020)

(1) is a function of one argument, of type `T`;
(2) is defined by `N` equations, one for each constructor of `T`;
(3) returns refinement expressions; and
(4) obviously terminates, often by structural recursion.

LiquidHaskell provides support for lifting Haskell functions that satisfy this description to measures automatically;[59] for example, we could have

```
len :: [a] -> Int
len []     = 0
len (_:xs) = 1 + len xs
{-@ measure len @-}

emp :: [a] -> Bool
emp []    = True
emp (_:_) = False
```

[59]The Haskell expressions defining the result of the function must be trivially convertible into refinement expressions.

```
{-@ measure emp @-}

fst :: (a,b) -> a
fst (x,_) = x
{-@ measure fst @-}
```

Here, we see three measures: the length of a list (available in LiquidHaskell by default at a more general type); whether or not a list is empty; and accessing the first component of a tuple (also available by default). This demonstrates that measures are actually quite flexible, and do not need to "measure" anything in the obvious sense.

While measures are flexible, they are not universal; any slightly complicated function – such as `(++)` or `map` – will quickly become immeasurable. Without the ability to refer to arbitrary Haskell functions in type signatures, our ability to verify code is incomplete; we can prove many properties of functions, but we are not capable of "arbitrary" verification.

9.4.3. Refinement reflection. Thankfully, recent work (Vazou et al., 2018b) has extended LiquidHaskell to be able to use Haskell functions within the logical refinements using a technique dubbed *refinement reflection*. The idea behind this technique is to treat the equations of a function definition as logical equalities in the refinement logic. Suppose we have the factorial function;

```
{-@ fact :: Nat -> Nat @-}
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
{-@ reflect fact @-}
```

It is then reasonable to take, as logical axioms,

$$\texttt{fact 0} = 1$$
$$\forall \texttt{n} : \texttt{Nat}.\ \texttt{fact n} = \texttt{n * fact (n-1)} \qquad \text{if } \texttt{n} \neq 0.$$

The reason we wish to use logical axioms here is that SMT solvers can reason about *uninterpreted* functions. If we add `fact` as an uninterpreted function to the logical environment, and then provide the two axioms above, we can get the SMT solver to effectively evaluate `fact`!

The way we encode these axioms is by automatically enhancing the refined type of `fact`. Instead of `Nat -> Nat`, where `Nat` is a LiquidHaskell type synonym for `{v:Int | v >= 0}`, it becomes

```
{-@ fact
      :: n:Nat
      -> { v:Nat
         |  v = fact n
         && (n = 0 ==> fact n = 1)
         && (n > 0 ==> fact n = n * fact (n-1)) } @-}
```

Now, at each application of `fact n`, LiquidHaskell can automatically (1) conclude that the result is equal to `fact n`; and (2) if `n` is known, use one of the implications to "evaluate" `fact`. If `n` is unknown, the implications will hang around until more information about `n` is known.

Thanks to the rich type of `fact`, we can now prove theorems about it, in a way we could not before. For example, we can prove that `fact 3 = 6`:

```
{-@ fact_3_is_6 :: {_:() | fact 3 = 6} @-}
fact_3_is_6 = ()
```

Yes, this is a sufficient proof! The SMT solver can string the facts about `fact` together, and conclude that

$$\begin{aligned}
\texttt{fact 3} &= \texttt{3 * fact 2} && \text{because } 3 > 0 \\
&= \texttt{3 * (2 * fact 1)} && \text{because } 2 > 0 \\
&= \texttt{3 * (2 * (1 * (fact 0)))} && \text{because } 1 > 0 \\
&= \texttt{3 * (2 * (1 * 1))} && \text{because } 0 = 1 \\
&= 6.
\end{aligned}$$

This uses the algorithm of *Proof By Logical Evaluation*, or *PLE*.

9.4.4. Termination and laziness. One important feature we see in LiquidHaskell is that, by default, it checks that every Haskell function is total. This is important for verification, as it means that our conclusions are actually theorems. Without termination, we cannot conclude that our theorems are actually theorems. For instance, we can define

```
{-@ evil :: () -> {false} @-}
evil :: () -> ()
evil u = evil u
```

where the refinement type `{p}` is short for `{_:() | p}`, and then prove anything we want simply by appeal to `evil ()`. Our theorems thus become merely partial correctness results. This check is why we needed to annotate, above, that `fact` operated only on `Nats`; without it, LiquidHaskell would have recognized that `fact` diverged on negative numbers.

In order to check termination, LiquidHaskell is more flexible than simply checking that functions are structurally recursive; instead, it can use a well-founded measure in order to check any argument for termination. We can write

```
{-@ func :: x:A -> y:B -> C / [m x, y] @-}
```

to check that the tuple `(m x, y)` is decreasing for an appropriate `m`.

However, if we only want partial correctness, then we can leave this check off, either globally with the `--no-termination` command-line argument or locally with the annotation `{-@ lazy func @-}`. When we do, we must contend with the fact that Haskell is nonstrict. Recalling `evil` above, suppose it were marked `lazy` and consider the function

```
{-@ not_okay :: {_:() | false} -> {1 = 0} @-}
not_okay :: () -> ()
not_okay _ = ()
```

This function looks permissible – doesn't it follow from the principle of explosion? However, consider the application `not_okay (evil ())`. Because Haskell is nonstrict, this is equal to `()`, and so it would be a *terminating* term that would prove that `1 = 0`. Instead, the permissible version of this function is

```
{-@ okay :: {_:() | false} -> {1 = 0} @-}
okay :: () -> ()
okay () = ()
```

Matching on the argument forces it to be evaluated, and ensures that our precondition has in fact been proven. We have to make sure that our arguments respect the partiality of partial correctness!

9.4.5. Example. In order to see these features in action, we can look at verifying well-scopedness for a simple lambda calculus. For the full code, see Section 9.4.6; the necessary excerpts are presented along the way.

We take as our object language a simple lambda calculus using de Bruijn indices for variable binding:

```
data Expr = Var Int
          | Lam Expr
          | App Expr Expr
          deriving (Eq, Ord, Show, Read)
{-@ Var :: Nat -> Expr @-}
```

We specify that the type of variables can only contain natural numbers. We then specify what it means for a lambda expression to be well-scoped, given the number of current binders:

```
{-@ wellScoped :: Nat -> e:Expr -> Bool / [size e] @-}
wellScoped :: Int -> Expr -> Bool
wellScoped vars (Var x)     = x < vars
wellScoped vars (Lam e)     = wellScoped (1+vars) e
wellScoped vars (App e1 e2) = wellScoped vars e1 &&
                              wellScoped vars e2
{-@ reflect wellScoped @-}
```

The termination annotation here is in terms of the obvious `size` measure; without the annotation, `wellScoped` will guess that the first argument is the termination measure instead.

Now consider the lifting operation, which takes a term and increments every binder above the given threshold. This is used to ensure that substitution is capture-avoiding.

```
{-@ lift :: Nat -> e:Expr -> Expr / [size e] @-}
lift :: Int -> Expr -> Expr
lift d (Var x)
```

```
  | x < d          = Var x
  | otherwise      = Var (x+1)
lift d (Lam e)     = Lam (lift (d+1) e)
lift d (App e1 e2) = App (lift d e1) (lift d e2)
{-@ reflect lift @-}
```

We want to prove that `liftWellScoped` preserves well-scopedness, but under one more binder than before – that if we know that `wellScoped k e`, then we can conclude that `wellScoped (1+k) (lift d e)`. We can state this as a theorem, and prove it:

```
{-@ liftWellScoped ::
      k:Nat ->
      d:Nat ->
      {e:Expr | wellScoped k e} ->
      {wellScoped (1+k) (lift d e)}
      / [size e] @-}
liftWellScoped :: Int -> Int -> Expr -> Proof
liftWellScoped _ d (Var x)
  | x < d     = trivial
  | otherwise = trivial
liftWellScoped k d (Lam e) =
  liftWellScoped (k+1) (d+1) e
liftWellScoped k d (App e1 e2) =
  liftWellScoped k d e1 &&&
  liftWellScoped k d e2
```

Here, `Proof` is a LiquidHaskell-provided synonym for the type `()`, and `trivial` is a synonym for the value `()`. The `(&&&)` function takes two `Proof` arguments and conjoins their refinements; the SMT solver and PLE does the rest of the work for us. We see that the pattern-matching and recursive structure of the code mirrors the structure of `lift`; we do not have to do any manual verification work other than to get the induction right.

The proof for substitution is much the same; again, the details are presented in Section 9.4.6.

```
{-@ LIQUID "--reflection" @-}
{-@ LIQUID "--ple"        @-}

import Prelude hiding ((!!))
import Language.Haskell.Liquid.ProofCombinators

{-@ (!!) :: xs:[a] -> {i:Nat | i < len xs} -> a @-}
(!!) :: [a] -> Int -> a
(x:_)  !! 0 = x
(_:xs) !! i = xs !! (i-1)
```

```
{-@ reflect !! @-}

data Expr = Var Int
          | Lam Expr
          | App Expr Expr
          deriving (Eq, Ord, Show, Read)
{-@ Var :: Nat -> Expr @-}

{-@ size :: Expr -> Nat @-}
size :: Expr -> Int
size (Var _)     = 0
size (Lam e)     = 1 + size e
size (App e1 e2) = 1 + size e1 + size e2
{-@ measure size @-}

{-@ wellScoped :: Nat -> e:Expr -> Bool / [size e] @-}
wellScoped :: Int -> Expr -> Bool
wellScoped vars (Var x)     = x < vars
wellScoped vars (Lam e)     = wellScoped (1+vars) e
wellScoped vars (App e1 e2) = wellScoped vars e1 &&
                              wellScoped vars e2
{-@ reflect wellScoped @-}

value :: Expr -> Bool
value (Lam _) = True
value _       = False
{-@ reflect value @-}

{-@ lift :: Nat -> e:Expr -> Expr / [size e] @-}
lift :: Int -> Expr -> Expr
lift d (Var x)
  | x < d          = Var x
  | otherwise      = Var (x+1)
lift d (Lam e)     = Lam (lift (d+1) e)
lift d (App e1 e2) = App (lift d e1) (lift d e2)
{-@ reflect lift @-}

{-@ subst :: Expr -> Nat -> Expr -> Expr @-}
subst :: Expr -> Int -> Expr -> Expr
subst (Var x)     y v
  | x <  y            = Var x
  | x == y            = v
```

```
  | otherwise         = Var (x-1)
subst (Lam e)     y v = Lam (subst e (y+1) (lift 0 v))
subst (App e1 e2) y v = App (subst e1 y v)
                            (subst e2 y v)
{-@ reflect subst @-}

{-@ liftWellScoped ::
      k:Nat ->
      d:Nat ->
      {e:Expr | wellScoped k e} ->
      {wellScoped (1+k) (lift d e)}
      / [size e] @-}
liftWellScoped :: Int -> Int -> Expr -> Proof
liftWellScoped _ d (Var x)
  | x < d     = trivial
  | otherwise = trivial
liftWellScoped k d (Lam e) =
  liftWellScoped (k+1) (d+1) e
liftWellScoped k d (App e1 e2) =
  liftWellScoped k d e1 &&&
  liftWellScoped k d e2

{-@ substWellScoped ::
      k:Nat ->
      {e:Expr  | wellScoped (k+1) e} ->
      {y:Nat   | y <= k} ->
      {e':Expr | wellScoped k e'} ->
      {wellScoped k (subst e y e')}
      / [size e] @-}
substWellScoped :: Int -> Expr -> Int -> Expr -> Proof
substWellScoped _ (Var x)     y _
  | x <  y    = trivial
  | x == y    = trivial
  | otherwise = trivial
substWellScoped k (Lam e)     y e' =
  liftWellScoped k 0 e' &&&
  substWellScoped (k+1) e (y+1) (lift 0 e')
substWellScoped k (App e1 e2) y e' =
  substWellScoped k e1 y e' &&&
  substWellScoped k e2 y e'
```

9.4.7. LiquidHaskell and `hs-to-coq`. Now that we have seen how LiquidHaskell looks when applied to examples that can fit in a paper, it is important to close out by looking at how LiquidHaskell can be applied to large examples. In particular, we focus on applications where we have done analogous verification work in `hs-to-coq`, to best see the similarity (although of course LiquidHaskell has been applied more widely). One early example of this is when Vazou, Rondon, and Jhala (2013) applied their then-latest improvements to LiquidHaskell to verify a range of modules: basic parametric functions; vectors with known domains and ranges; both textbook and optimized sorting algorithms; a splay set from `llrbtree` (Yamamoto, 2012)[60]; and finally, the implementation of `Map`s from `containers`, which was the largest module they verified by an order of magnitude (in fact, it was up over 80% of the total lines of code they verified). This last we also verified (though a different version) with `hs-to-coq`, as we discussed in Chapter 6, providing a key point of comparison.

Vazou et al. (2013) found many things in common with what we found: specifying and verifying top-level functions was often simpler than the specifying and verifying the auxiliary functions they depend upon; code modification was occasionally necessary to allow the verification to go through; and they were able to verify the top-level functions in the implementation of `Map`. Their specification had many similarities to ours: we both used unbounded integers, rather than dealing with machine words, and we both left functions such as `showTree` unverified. However, our verification provided a richer specification; while Vazou et al. (2013) verified only that the trees were binary search trees, we also verified that each operation was semantically correct and that the trees were *weight-balanced* as well as binary search trees.

In a similarly related vein, though at a less real-world scale, Vazou, Breitner, Kunkel, Horn, and Hutton (2018a) verified examples from *Programming in Haskell* (Hutton, 2016). This was a demonstration of LiquidHaskell's excellent support for equational reasoning, showing how LiquidHaskell can make proofs look *nice*, something out of reach for Coq. By defining appropriate combinators, proofs in LiquidHaskell can look remarkably like the sort of proofs by equational reasoning that Haskell programmers, including Hutton, like to employ. In particular, in Section 5, Vazou et al. verify "Hutton's razor", the simple compiler from an expression-based language to a stack machine that we also verified (Section 5.2). They too had to face the question of totality, since the `exec` function that evaluates the compiler is partial. LiquidHaskell had an option here we did not: they could have used refinement types to transparently restrict the domain of `exec` to well-formed code-stack pairs. However, this would have involved a complicated binary predicate, and so they chose to resolve the dilemma the same way we did: by making the code explicitly use `Maybe`. The structure of their proof is the same as ours: the same top-level theorem, that `exec (comp e) [] == Just [eval e]`; the same inductive form of the theorem, that `exec (comp e) s == Just (eval e : s)`; and the same distributivity lemma, that `exec (c ++ d) s == exec c s >>= exec d` (although here, we used different variable names and wrote our bind in the other direction, as `=<<`).

[60]This appears to be the correct package and version, but they do not provide an explicit reference.

Finally more recent work has brought LiquidHaskell to type classes (Liu, Parker, Redmond, Kuper, Hicks, and Vazou, 2020). A key component of `hs-to-coq` has been our provision of laws for type classes such as `Eq`, `Ord`, `Semigroup`, `Monoid`, `Functor`, `Applicative`, and `Monad`, as I discussed in Section 5.1. However, until recently, LiquidHaskell could not provide laws for type classes. It could verify properties of individual instances, but not apply general refinements to the types of class methods themselves. Once Liu et al. (2020) designed implemented support for this, they both: (1) brought it to bear on the standard type classes `Semigroup`, `Monoid`, `Functor`, `Applicative`, and `Monad`; and (2) used it to verify an implementation of replicated data types. As with our approach, when providing law-bearing versions of type classes, they provide a separate set of type classes; they prefix them with `V` (presumably for verified) instead of suffixing them with `Laws`, thus getting type classes such as `VSemigroup`.

One wrinkle that both Liu et al. and `hs-to-coq` had to handle was *type class coherence*. In Haskell, type class resolution is *coherent*, meaning that all solutions of a type class constraint are the same values. In Coq, this is not the case, and so when we have the three requirements that

(1) an instance of `Applicative` is also an instance of `Functor`;
(2) an instance of `FunctorLaws` is also an instance of `Functor`; and
(3) an instance of `ApplicativeLaws` is also an instance of both `Functor` and `FunctorLaws`,

we naïvely would get two different instances of `Functor` for the same type. When we built the Coq libraries for `hs-to-coq`, we had to avoid that manually; when extending LiquidHaskell, they leverage Haskell's constraint solver. However, Haskellers can use the `{-# INCOHERENT #-}` pragma to violate coherence for specific instances, and for a verification tool like LiquidHaskell, this cannot be ignored. Thus, they extend LiquidHaskell to *check* that the instance dictionaries they use are coherent, thereby retaining soundness.

The list of type classes that Liu et al. verify notably left out `Eq` and `Ord`. It turns out that these type classes, as well as `Num`, were already special-cased in LiquidHaskell, and linked directly to the appropriate SMT solver primitives; this is more powerful that the refinement types they would have, since the SMT solver knows how to reason about those theories. Since making that change would both lose that property and break existing proofs, Liu et al. left this handling alone, looking to build a hybrid SMT–type class refinement approach in the future.

Using these techniques, in addition to verifying the standard type classes and instances, Liu et al. verified a library of *replicated data types* (RDTs, or VRDTs when verified), which are data types that can have replicas distributed across multiple computers, receive unordered update operations, and converge to the same result. They then used these to implement two applications: "a *shared event planner* and a *collaborative text editor*." The VRDT development was 5536 lines of code, and featured something familiar: an implementation of `Data.Map`. However, in order to verify that module, Liu et al. had to *redefine* it so that they could verify the properties they needed. In particular, they verified that map keys were appropriately sorted, and not the more comprehensive specification that we verified (Chapter 6). This was not

the only code that they had to redefine in order to hang refinements on, but merely one example that we recognize well. Once they had this version of `Data.Map`, they were able to link it to the SMT solver's theory of sets, gaining a powerful library of verification (although they experienced some growing pains, having difficulty linking the SMT theories and their type class refinements).

9.4.8. LiquidHaskell in review. We have thus seen how LiquidHaskell, our closest neighbor in the space of verifying existing, realistic Haskell programs, is a powerful and effective tool for carrying out verification. By using SMT solvers and annotating existing Haskell programs with refinement types, their verification can be even more transparent and smooth than `hs-to-coq` can be. However, by working with Coq, we get the ability to leverage an existing infrastructure and ecosystem devoted to verification, and can use the full power of the Calculus of (Co-)Inductive Constructions to verify programs rather than solely the language of an SMT solver. The tradeoffs that LiquidHaskell and `hs-to-coq` make do not position either as clearly a winner in the space, and I look forward to seeing what both they and we will do next.

9.5. A verified functional language: CakeML [61]

One major application of `hs-to-coq`, as we saw in GHC Chapter 7, was verifying parts of GHC. There has been a lot of work done on verifying compilers; for instance, CompCert is a fully formally verified C compiler written in Coq (Leroy, 2009). Within this field, the CakeML project (Kumar et al., 2014; Tan et al., 2016) has particular relevance to `hs-to-coq`. CakeML is a verified compiler and REPL for an ML dialect, also called CakeML. Its particular connection to `hs-to-coq` is that both leverage translation in their compiler verification; the CakeML implementation is a masterful, tightly-woven application of translation from HOL4 to CakeML. The translation is different, in that the original code is written in HOL4 and this is where the verification works takes place; furthermore, their translation is certificate-producing (Myreen and Owens, 2012), so they do not need to trust it the way that we must trust `hs-to-coq`. This also means that they do not need to worry about the legibility of the translation. On the other hand, there are similarities: the use of verification on a compiler for a functional language, the verification work happening within a theorem prover, and the fact that they get both a CakeML and HOL4 (analogous to Haskell and Coq in our setting) implementation of the compiler.

There are two versions of the CakeML compiler: a 2014 version, which provided a verified REPL and focused on ease of verification (Kumar et al., 2014); and a 2016 version, built upon the former, which focused on a fully-featured source language and more efficient code generation (Tan et al., 2016). (And work has not stopped (Abrahamsson, Åman Pohjola, Fox, Gómez-Londoño, Kanabar, Kumar, Lööw, Myreen, Norrish, Owens, Sewell, Syeda, Tan, Tomandl, and other contributors, 2020a) – but the present model for the project is in the vein of the 2016 paper.) An important key feature of CakeML is that the compiler is *compiled with itself* (i.e., it is *bootstrapped*),

[61] This text is originally from my WPE II.

meaning that we not only know that the source of the compiler is correct, but that the actual executable is correct as well.

Both versions of CakeML wind up with a high-level correctness theorem that says, to paraphrase, roughly the following:

> If we take a CakeML expression (2014)/program (2016), compile it, and run the result, either:
> (1) the CakeML evaluates successfully and so does the compiled output, producing the same behavior;
> (2) the CakeML crashes and so does the compiled output, producing the same result before that point; or
> (3) the compiled code behaves like the first part of the CakeML program, and then crashes early with an out-of-memory error.
>
> None of this holds if anything else can interfere with the heap.

Of course, the devil is in the details.

The workflow of producing the CakeML compiler is as follows.

(1) The compiler is written and verified in HOL4.
(2) CakeML code is synthesized automatically from the HOL4 code, using a certificate-producing algorithm (Myreen and Owens, 2012). This produces a CakeML implementation of the compiler.
(3) The HOL4 implementation of the compiler is run, inside HOL4, on the generated CakeML source code for the compiler. This produces, again in HOL4, a sequence of machine instructions that implements the compiler.
(4) Thanks to the compiler correctness theorem in HOL4, we know that the sequence of machine instructions does the same thing as the input CakeML source code.
(5) Thanks again to the compiler correctness theorem, but this time the version generated with the proof-producing synthesis in step 2, we know that the thing the input CakeML source code did was correctly compile CakeML.
(6) Therefore, the sequence of machine instructions is a correct CakeML compiler.

At this point, we could run the machine instructions in an executable and provide the generated CakeML code from step 2 as input if we wanted. This whole workflow is also drawn out diagrammatically in Figure 9.2.

In many ways, the construction of the CakeML compiler is entirely standard. The 2014 version consists of four stages:

(1) Parsing;
(2) Type inference;
(3) Conversion to a functional IL; and
(4) Conversion to a bytecode.

In this version, the machine code is generated with some automation, but is specifically done for the REPL; the compiler itself only goes as far as bytecode, and it is this that is bootstrapped. The executable REPL, which is specifically for x86-64, is a bespoke artifact generated from the bootstrapped bytecode.

In the 2016 version of CakeML, there are instead *twelve* intermediate languages, multiple target machines, and thirty different passes, from parsing to machine code generation. This is, in fact, even more standard than the 2014 version, and indeed

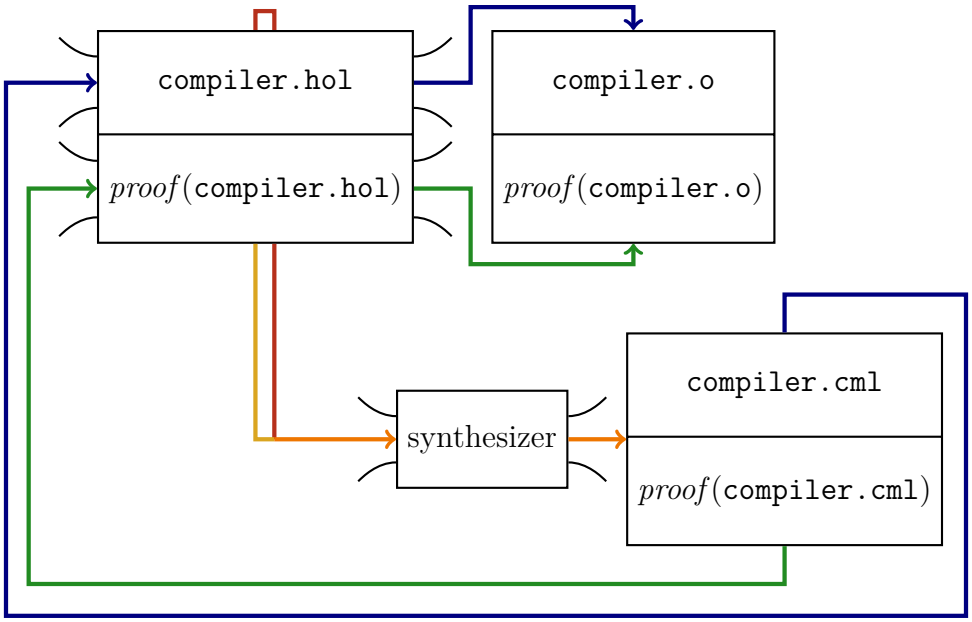

FIGURE 9.2. How to get to CakeML. Arrows represent the flow of values as inputs and outputs; the "chutes" indicate that the attached block is being applied to the arrow-fed input to produce the arrow-emitted output. Plain blocks without chutes, such as `compiler.cml`, are never applied, and are only static data.

gives better performance. Here, the bootstrapping goes all the way to the bare metal and back up to the CakeML.

While the internal details of all these passes are surely interesting, they are not what we are here for. We are here for the verification. In particular, the verification of the REPL in the 2014 version of the paper. The formalization of a REPL, as opposed to a full compiler, is one of the most unusual features of this CakeML formalization.

The specification of the REPL, REPL_s, is a HOL relation of type bool list → string → repl_result → bool. It relates three things. Not in order, they are:

(1) The input string, which will hopefully be parsable as a sequence of declarations.
(2) A list of booleans indicating the presence of type errors in those declarations.
(3) The result: a list of printed outputs, followed by "terminates" or "diverges".

As a REPL, the idea is that each declaration will either print out a result value, print out a reported error, or diverge entirely. The booleans are necessary because the type inferencer has only been proven to be sound, but not complete; if the semantics tried to execute a typeless program, it would get stuck.

This specification is connected both to the operational semantics – which is available in both big- and small-step styles – and to an HOL implementation of the REPL, REPL_i. This is a total but *uncomputable* function of type string → repl_result; it is uncomputable because it must end the list of results with "diverges" if and only if

running the input will diverge. This is, of course, undecidable (Turing, 1937). The ability to work with functions like this is one of the powers of HOL.

The function REPL_i – or, more precisely, its subcomponent REPL_i_step, the function which converts an input string to bytecode – is synthesized into CakeML code, which is then put through the whole involved bootstrapping process. The key lemma connecting REPL_i and REPL_s says that for all inputs s, there exists a list e such that $\mathsf{REPL}_s\ e\ s\ (\mathsf{REPL}_i\ s)$. Combined with the bootstrapping process, this gives us our correct byte code core of the REPL. This is then translated to x86-64 machine code, and placed within a main loop, garbage collector, and libraries all additionally constructed from synthesized machine code with proofs. The result is a verified REPL whose correctness follows from combining all of the various proofs about code that have been preserved, synthesized, translated, or even proved manually.

The 2016 formulation (Tan et al., 2016) helps close some of the gaps present above – although it lacks a REPL, the type inferencer has been proven to be both sound *and* complete, the compilation story goes all the way down to machine code, and things like the garbage collector are now accounted for. It does still rely on assumptions that any FFI calls are correct and do not hammer on its memory, but this is a much more mild assumption.

9.5.1. Verifying programs? Because the CakeML compiler is verified against a semantics, it is possible to use CakeML to write verified programs. However, this is not the intended use of CakeML; instead, the project's website says that "[u]sually, we recommend that verified CakeML code is produced via synthesis using frontend 1 [the HOL-to-CakeML synthesizer]." (Abrahamsson, Åman Pohjola, Fox, Gómez-Londoño, Kanabar, Kumar, Lööw, Myreen, Norrish, Owens, Sewell, Syeda, Tan, Tomandl, and other contributors, 2020b) While it does acknowledge that "in some cases it is more convenient to do Hoare-style reasoning in the separation logic of CFML [Characteristic Formulae for ML, specifically a version adapted to CakeML]" (ibid.), the general approach to verifying CakeML programs is to simply use a theorem prover and generate verified code from the result – a more powerful form of extraction.

CHAPTER 10

Conclusions and Future Work

In this dissertation, I have presented `hs-to-coq`, a novel tool for faithfully translating Haskell programs into legible Coq code. We have seen the design of the tool itself, the design of the edit language that it supports, and the methodology for how to use it. And we have seen how `hs-to-coq` can be brought to bear on problems ranging from the `base` library, through `containers`, all the way to GHC itself. Finally, we saw other work in the same space, and how `hs-to-coq` has advantages and disadvantages when contrasted with these other techniques.

At the beginning of this project, we had hoped to build a tool that could translate Haskell type definitions into Coq, and use that to specify parts of GHC. Very quickly, I realized `hs-to-coq` had the potential to apply to terms as well, and the result gradually became what we have seen here. Yet GHC proved a more slippery target than we had hoped, in part due to the expansion of our goals for `hs-to-coq`, and though we started with small pieces such as `Bag`s early on, it took us some time before we were able to translate realistic portions of GHC. Happily, this is something we can now do, and verification work on GHC is continuing to expand.

The design of the edit language was not originally part of our concept of `hs-to-coq`, but it became apparent that it was the beating heart of `hs-to-coq`, the feature without which we could not do anything we wanted. The edit language enabled us to use `hs-to-coq` anywhere on a spectrum from "nearly direct correspondence", as with our initial work which we saw in Chapter 5, to "creating a model of the program that is more amenable to verification", as with our verification of GHC which we saw in Chapter 7. This range, combined with the machine-readable nature of the edit files, gave rise to our concept of the *mechanized formalization gap*, a novel contribution of `hs-to-coq`. There is always a formalization gap in any verified in system; in `hs-to-coq`, the domain-specific portion of that gap (beyond `hs-to-coq` itself) is recorded, having been written down as edits.

10.1. Edits: a retrospective

The work we have done in all of our projects required edits, ranging from those that are minimally invasive (such as `rename type`) to those that are dramatic transformations (such as `inline mutual` or `skip constructor`). We chose which edits to add as part of a feedback-driven design process, adding edits as they were needed by the translations and proofs. This is why we had a more modest, less invasive set of edits when doing our initial work (Chapter 5); why we added more complex edits when verifying `containers` (Chapter 6); and why we finally built up to our current set of edits, including those that alter the code the most, when verifying portions of GHC (Chapter 7). How do these fit into the verification process?

The status quo, with Coq verification, is to think of the verification project as containing three pieces:

(1) The domain objects to be verified;
(2) The theorems about (equivalently, the specifications of) these objects; and
(3) The proofs of these theorems.

These steps are distinct: in order to successfully complete a verification project, you first have to acquire a Coq model of your domain, either by writing it or using an existing one (an example of the latter being the natural numbers when verifying a number-theoretic property); you then have to settle on the desired specification of these objects; and finally, you have to carry out the proof. There is often feedback between these steps – for instance, proofs requiring intermediate lemmas, which then have their own proofs – but they represent three different conceptual activities, each of which is represented in its own domain in Coq (roughly speaking, (1) `Definition`s in `Type`, (2) `Theorem`s in `Prop`, and (3) `Proof`s in Ltac). And each of these pieces is challenging in its own right (with the possible exception of (1) if you are working with an existing object such as the natural numbers).

Using `hs-to-coq` adds a fourth piece:

(4) The formalization gap between the Haskell domain objects and the Coq model of the domain objects.

This gap is represented, of course, in the edit files written for the particular development. Because edit files can be written for every project, and because they are usually necessary, considering the precise details of the formalization is a mandatory part of using `hs-to-coq`, and it is a new source of difficulties during a verification project. Defining the formalization gap is tricky, just like producing the other three pieces of Coq projects; at the time of writing, our verification of `containers` contains 672 (nonblank, noncomment) lines of edits, and our verification of GHC contains 1400–1804 lines of edits (depending on exactly which per-module edits we include, as the translation is still in progress). However, we believe that the benefits of this approach are worth it; as we can recall from Section 1.2, these benefits are that edit files enable us to

(1) verify existing Haskell code;
(2) never edit Haskell code by hand;
(3) get a mechanized formalization gap; and
(4) work anywhere on the spectrum from confidence to perfection.

The volume of edits for `containers` and GHC – over half of all the edits in our repository, which weigh in at 4142 lines in total – comes with two quantitative lessons. On the one hand, edits are challenging to write; we have written literally thousands of lines of them. On the other hand, edits are simpler than the programs they apply to; for both `containers` and GHC, the edits are an order of magnitude smaller than the translated and verified code. This juxtaposition highlights another reason we believe that edit files are worth their weight when using `hs-to-coq` to simplify the semantics of the translated Coq model, as in the GHC case. While writing edits is an additional cost incurred by using `hs-to-coq`, a relatively smaller amount of time spent on edits can pay tremendous dividends in the amount of time saved during

verification. In the case of GHC, verification would simply not have been possible without our suite of edits; in general, the order of magnitude size difference, along with our experience during the verification process, points to the cost of writing edits as being meaningfully less than the amount of work they save.

We found that the edits we needed to write for a verification project were always highly influenced by inspecting the result of translation; this was especially true at first, when we still didn't fully understand how to write effective edit files, but the bespoke nature of the edits for each project meant that in no case could we design the formalization gap from first principles. The process of verification was a loop: translate Haskell, find problems in the resulting Coq code, experiment to determine what transformations would be necessary, write those transformations down as edits, and repeat. Finding problems on the Coq side could either happen due to a failed compilation or later due to difficulties during specification design or proof writing; the former was more likely to happen during the initial phases of translating and verifying any individual piece of code, and the latter to happen later, once working with the core of a translation. Determining the edits sometimes involved manually tweaking the Coq code to determine what changes would be necessary (e.g., for a `rewrite` edit), but the autogeneration of the Coq output meant that we could not keep those changes and so were unable to take the shortcut of "just editing code", as desired.

For much of the early part of our work, there was an additional step to this loop: after determining what transformations would be necessary, we often had to extend `hs-to-coq` with a new edit that could perform it. Early projects found me doing this almost constantly, as this is where the edit language came from; later on, our development process could often proceed without adding new edits to `hs-to-coq`, but the need to add new ones still cropped up regularly. It was only by the time we began to work on translating and verifying parts of GHC – fairly late in `hs-to-coq`'s life – that the edit language truly began to stabilize.

10.2. Evaluating the edit language

The design process of the edit language produced a set of edits that can be thought of us "two-sided" in the sense of DeepSpec. Recall that "two-sided" means "connected to both implementations and clients" (Appel et al., 2017). The edit language is not a specification, but plays a similar role; when thinking about two-sidedness in this context, we take it to mean "useful for both translation and verification". And indeed, the feedback driving the design of the edit language came from both the translation and verification halves. Some edits were more important for one than the other: for example, `skip constructor` was necessary when translating GHC, in order to remove constructors that trigger unsupported features such as non-strictly-positive recursion in data types. While those changes also eased verification, our theorems could alternatively (albeit significantly less elegantly) have been stated to ignore the skipped cases. On the other hand, the `inline mutual` edit had to be useful for both translation, which meant producing correctly mutually-recursive code, as well as verification, which meant producing code that was pleasant to work with and similar to the original Haskell.

Attribute	*DeepSpec*	*Edits*	*Success?*
Rich	"Describing complex component behaviors in detail"	Capable of expressing arbitrary complex code transformations	✓
Two-sided	"Connected to both implementations and clients"	Useful for both translations and proofs	✓✓
Formal	"Written in a mathematical notation with clear semantics"	A machine-readable language with clear semantics	–
Live	"Connected via machine-checkable proofs to the implementation and client code"	Connects the Haskell to the Coq automatically via `hs-to-coq`	✓✓

FIGURE 10.1. Are edits a deep language? Evaluating how the attributes of a deep specification (Appel et al., 2017) correspond to the design and features of the edit language.

In fact, the whole DeepSpec framework can be reinterpreted from this new perspective as a guide to understanding how we want to think about edits. Recall that the four attributes of a deep specification are that it is (1) rich, (2) formal, (3) two-sided, and (4) live (Appel et al., 2017). We want our edit language to also be "deep" in this sense. What does that mean for us? In Figure 10.1, we draw analogies between what these four attributes mean for specifications and what they mean for us.

10.2.1. Are edits rich? Richness is a measure of how expressive specifications are. For edits, the corresponding richness is in the code transformations that can be expressed. We have found the collection of edits we've come up with to be sufficiently rich – while `rewrite`, `add`, and `skip` are the most emblematic of this, since they can transform code in truly unconstrained ways, we have found our collection of edits to form a useful vocabulary. Edits like `skip type` and `inline mutual` express concrete transformations we want to apply, and can be used widely across different modules and even projects.

That said, the richness of the edit vocabulary is missing one component: abstraction. Without a facility for edit abstraction, changes to the edit language need to be made within `hs-to-coq`. Adding `inline mutual` or `set type` could only be done by writing new Haskell transformations on the AST, and not from within the edit language itself.

During the verification of GHC, we often saw that we could get very far without extending `hs-to-coq`. This was a new experience – our first verification projects, up through `containers`, involved producing new edits from the get-go. While we knew working on GHC would also involve this, the balance had noticeably shifted; the features we had added to the edit language while working on `containers` had dramatically increased the usability of the edit language. This also meant that the

new edits we added were targeted precisely at pain points in the translation and verification processes.

All in all, the edit language is satisfyingly rich; that said, individual complex edits (e.g., `inline mutual`) must be added by hand to `hs-to-coq`, so we cannot ever say that we are "completely rich" and can stop adding edits.

10.2.2. Are edits two-sided? For a specification to be two-sided, it has to connect to both the artifact being verified and the proof that the artifact is correct. We discussed what this means for `hs-to-coq` earlier, but in some ways, edits have three sides: first, `hs-to-coq` (are they implementable?); second, the translation (do they apply to the artifact?); third, the proof (do they produce verifiable Coq?). As I discussed above, our feedback-driven process of working on the proofs and `hs-to-coq` simultaneously means our edits are very much two- (or three-?) sided.

10.2.3. Are edits formal? Formality is a measure of how "mathematical" specifications are, in a certain sense – how precise and unambiguous they are, which usually corresponds to using mathematical notation. Since edits are a programming language, they automatically have *a* semantics – it's the Haskell code of `hs-to-coq` that describes the transformation. However, we do not go further than this; there is no "core calculus" of edits which we reason about, and while we provide documentation for all the edits (see Chapter 4 and https://hs-to-coq.readthedocs.io/), this documentation is not itself formal (or deep in general).

Additionally, since our edits grew organically, we don't have a notion of why *this* set of edits is the one to have. The two-sidedness of edits tells us that we have *a* useful set of edits. But our set is not privileged in any sense – it is not minimal, or theoretically nice, or orthogonal. It would be an interesting project to look at the collection of edits we have accrued over the years and think about how to capture them as a language that was designed all at once. Until then, our edits are mechanized and documented, but not truly *formal*.

10.2.4. Are edits live? A specification is live when it is constantly being validated. A specification written on paper can get out of date with respect to either the implementation being verified or the clients of the specification; a live Coq specification cannot, as it's validated every time that it's compiled. This is the hardest property to consider when looking at the edit language, as the language design itself doesn't directly touch either the Haskell code we translate or the Coq code we verify. I believe the closest analog of "being live" for the edit language is that it should *enable* a live translation – it should allow for a machine-run connection between the Haskell artifact, the Coq translation, and the Coq proofs. And `hs-to-coq` – in concert with some Makefiles – provides exactly this by design.

There is certainly room to refine this notion of being live even further. The Makefiles we use to build our work are hand-crafted and fragile, and `hs-to-coq` could track more dependency information on its own. And we have not yet tested how robust edit files are when we apply them to a project that is changing over time. But the key piece of a mechanized formalization gap is that we support this sort of

live connection, and this has been a key and successful part of `hs-to-coq` from the beginning.

10.2.5. Are edits deep enough? So, having gone through these four attributes, is the edit language deep, or at least deep enough? I believe it is – we could not have worked with a project at the scale of GHC without it. One of the biggest weakness of the edit language as it stands is the somewhat ad-hoc nature of our selection of edits: we added `redefine`, and then separately `add` and `skip`; we added individual complex transformations, such as `inline mutual`, while also supporting the customizable `rewrite` edit; and so on. Future work could take the current edit language and refine it down into something more streamlined, having learned the lessons we took from designing it this first time. But the key attributes – individual, named code transformations; being expressive enough to change the semantics of code in arbitrary ways; and supporting a mechanized formalization gap – coupled with our two-sided feedback-driven development process are crucial enablers of the style of verification we outline here. Even in this refined edit language, I believe this key framework would remain.

10.3. Future work

The story of `hs-to-coq` doesn't end here. This work was fundamentally a joint effort, and the `hs-to-coq` team has together built `hs-to-coq` into a powerful tool for verifying Haskell code. The features it currently has and the applications to which it has so far been put are only the tip of the iceberg. There is always more work to be done (as even a cursory glance at our GitHub issues page[62] will tell you!).

One key place to consider what could be done is to look at the edit language, which forms the backbone of `hs-to-coq` by enabling us to provide a mechanized formalization gap. As is the case for designing a good specification, designing a good formalization gap is an art, and one that our work has just begun to scratch the surface of. The lessons we have learned have taught us a great deal about what to put in effective edit files: the sorts of code we want to skip; how to handle tricky termination arguments; aiming for confidence instead of perfection for large projects; the first "design patterns" such as justified edits (Section 7.7); and more. At the same time, we have also learned some of the features that make a good edit *language* that can underly this art: support for flexible edits, like `rewrite` and `set type`; edits that directly address common pain points, like `termination`; edits that perform complex transformations that show up repeatedly, like `inline mutual`; and edits that perform basic building-block operation but aren't flashy, like `skip` and `add`.

As this is the first work to introduce a mechanized formalization gap, however, we don't believe we have all the answers. Looking forward, there are multiple possible ways to potentially improve or extend the edit language; it is not clear which approaches will prove the most fruitful, but the ideas that we have already seen and the problems we have already had provide signs that point towards promising ideas. The clearest avenue here (besides fixing bugs) is adding more edits, an endlessly possible task;

[62] https://github.com/plclub/hs-to-coq/issues

however, there are also other, significantly more far-reaching changes that could be made as well.

The most direct possibility is to completely redesign the edit language to have a principled core. The edit language has clearly accreted over time, and was not designed to have a clean core or precise semantics: as an example of the former, we have the `redefine` edit, which could be replaced by `skip` plus `add` or `add type`; as an example of the latter, we have complex code transformations such as `inline mutual` or `collapse let`. This means that understanding the edit language requires understanding 34 distinct moving parts, and although they have certain kinds of internal symmetry (e.g., the various edits to skip code all behave similarly and have similar names: `skip`, `skip constructor`, `skip class`, ...), there is no fundamental connection going on behind the scenes. This would also potentially simplify the implementation of `hs-to-coq`; a more uniform core semantics could perhaps be reflected in a more uniform implementation strategy.

Relatedly, we also might consider how to give the edit language more features from full-fledged languages. In particular, we might consider how to support *abstraction* within the edit language. This could potentially enable us to define new edits within edit files, enabling users to define novel edits without extending `hs-to-coq`. It is not clear what the correct semantics for such an approach would be, or what other ways we might give the edit language other "full-fledged" features; developing a precise semantics could shed light on this approach. One avenue would be to conceptualize the edit language as a domain-specific language for tree transformations; however, the high-level nature of the edits means that this may be too detailed to smoothly support the kinds of transformations we want to express.

Looking in a different direction, we could consider providing different ways of working with our existing edits, reflecting challenges we faced in our current verification efforts. As is often the case, this is well-reflected in considering design patterns that might benefit from more formal support in the language. One pattern we came across frequently is the distinction between "global" and "local" edits (also discussed in Section 4.4). Certain individual edits, such as `rename type GHC.Base.Maybe = option` or `skip class GHC.Show.Show`, need to be present when translating every module in a program; other edits, such as `skip Core.tyCoFVsOfType` or `rename type GHC.Num.Int = nat`, are only needed within the translation of one module. As we can see from these examples, this distinction is not simply one of "which of the 34 edits is this"; while `skip class` always needs to be global and `skip` can always be local, `rename type` can be either, depending on the desired semantics. In fact, the specific edit `rename type GHC.Num.Int = nat` could be either local or global, depending on whether this change was intended to control computation within just one module or to specify a default interpretation! Thus, determining whether edits are global or local is something that needs to be controlled by the user, at least sometimes. We currently handle this process by using one file for our global edits and a bevy of files for our local edits (one per module), but this is built on top of complex Makefile behavior and not part of the edit language; we also do propagate some information between translated modules, but not enough

to propagate all global edit information. Reifying these distinctions in a way that was visible to both the user and to `hs-to-coq` itself could help smooth over this process.

Another design pattern with fertile ground for exploration is the *justified edits* technique, discussed in Section 7.7. This technique allows us to increase our confidence in our translation by providing *formal* Coq proofs that are *informally* linked to edits via comments, where the proofs indicate that the rewritings performed by those edits do not change the behavior. This technique is one of the most recent developments for writing edit files, so it has not yet been fully explored. Two potential directions for future developments here would be (1) providing a way to classify edits as justifiable or not while extending this pattern to more edits; and (2) replacing the informal, comment-based form of justifications with a live (in the sense of DeepSpec), mechanized form of justifications that `hs-to-coq` understands. Both of these possibilities would allow us to increase the confidence that we gained from using this design pattern (or rather, after these changes, this feature); however, both of these require navigating a complex solution space to determine the correct design. Each has potential benefits, but is complicated by clear challenges in setting up the design.

How to answer the first question – whether an edit is self-justifying (such as `inline mutual`, which is defined not to change semantics), justifiable with a proof (such as `rewrite`, as in all our current examples), or unjustifiable (such as `axiomatize`, which is intended to change the meaning of the translation) – is already unclear; consider the example of `collectNBinders`, which closes out Section 7.7.3 and has a justification theorem which is a *disjunction* of an equality and a conjunction of two instances of the custom inductive type `panicked`. This demonstrates that determining the meaning of justification for a `rewrite` edit, even one intended to be semantics-preserving, is nonobvious. We could also imagine wanting to verify that a transformation is "close enough" in other ways (not just "up to `panicked`"); for instance, close approximations of the same real-valued function, or simplifications that make assumptions about the input domain.

Extending these questions to other edits exposes other challenges. For instance, consider the `skip` edit. In one sense, `skip` is self-justifying, since `skip` does not delete any uses of the skipped value; this means that Coq code that compiles after a `skip` must not have referred to the skipped definition. In another sense, `skip` is unjustifiable, because it deletes an entire definition! The example of `skip` can be used to highlight another challenge with this approach: suppose that we decided that `skip M.x` alone is always self-justifying, because if the result compiles no translated definitions can refer to `M.x`; similarly, suppose we say that `add` edits such as `add M Definition M.x := false.` alone are always self-justifying, because Coq cannot shadow names at module scope; this means that if the result compiles there must not have been an existing `M.x` and so nothing could have referred to it. But then, if we started with the definition `x = True` in `Mod.hs`, the result of these two edits would be to invert `M.x` by effectively redefining it in-place, making these edits unjustified. It thus seems that the notion of justification is not compositional, and may require reasoning about the interaction between multiple edits.

Looking instead at the question of how to make the notion of justification live, we can again consider `collectNBinders` and its need for disjunction; simply reasoning

about equality is not enough. It might be possible have an equality in the edit file that is both used to rewrite the translated code and also copied to the output as a theorem; however, we then need to provide a way to prove the theorem, which can only be done after the translation is complete. This could perhaps be handled in the same manner as the `obligations` edit, although the differences between the proofs could make that harder. This could be enough to provide a potentially useful first step, but addressing other forms of "close enough to identical" behavior would remain a challenge.

Moving away from the specifics of the edit language, another direction for extending this work would be to improve the design of `hs-to-coq`'s translation. One future project in this vein that could pay dividends would be converting `hs-to-coq` to use the Equations package (Sozeau and Mangin, 2019) to produce more Haskell-like output and handle mutual recursion more easily.

Another angle of future work is simply the further application of `hs-to-coq` to larger projects. While this is not unique to the `hs-to-coq` team, we have our own designs: we hope to extend to verifying effectful code, and to verify even more of GHC. We have proven basic results about monadic functions and verified core pieces of GHC, but there is more to do in both cases. In the former case, we would like to be able to reason about `IO`. In the latter case, it would be wonderful to see our translation and verification of GHC extended to results about the semantic correctness of optimization passes, or even the correctness of GHC's internal "Core lint" pass, which checks the types of Core programs when debugging GHC. Outside of our team, there is a project in the Programming Languages and Verification Group at MIT CSAIL to specify the RISC-V architecture that generates its model using `hs-to-coq`; their work is available at https://github.com/mit-plv/riscv-coq.

One final set of exciting possibilities would be to attempt to connect `hs-to-coq` output with existing Coq formalisms. Again, one prospect is to do this with our translation of GHC; there exists ongoing work on producing a mechanized semantics of System FC (or its new dependently-typed variant), and it would be wonderful to align our semantics of Core with this mechanized semantics.

10.4. `hs-to-coq`

As demonstrated by the results in this thesis, `hs-to-coq` is readily usable today. The code is available online at https://github.com/plclub/hs-to-coq, and the documentation is available online at https://hs-to-coq.readthedocs.io/. There, you can see the complete code that backs this thesis, as well as all the work that has happened since these words were written. The next time you find yourself wanting to verify Haskell code or to retain a mechanized formalization gap, I hope you will turn to `hs-to-coq` and join us!

Bibliography

Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. Verifying Haskell programs using constructive type theory. In Daan Leijen, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2005, Tallinn, Estonia, September 30, 2005*, pages 62–73. ACM, 2005. ISBN 1-59593-071-X. DOI: 10.1145/1088348.1088355. URL https://doi.org/10.1145/1088348.1088355. Citations: pp. 32, 196, 196, 196, 197, 197, and 197.

Oskar Abrahamsson (`oskarabrahamsson`), Johannes Åman Pohjola (`IlmariReissumies`), Anthony Fox (`acjf3`), Alejandro Gómez-Londoño (`agomezl`), Hrutvik Kanabar, Ramana Kumar (`xrchz`), Andreas Lööw (`AndreasLoow`), Magnus Myreen (`myreen`), Michael Norrish (`mn200`), Scott Owens (`SOwens`), Thomas Sewell (`talsewell`), Hira Syeda (`hirataqdees`), Yong Kiam Tan (`tanyongkiam`), Timotej Tomandl, and other contributors. CakeML: A verified implementation of ML (GitHub repository). August 5, 2020a. URL https://github.com/CakeML/cakeml. Date is of last access (commit `c89761b7ef`). Contributors are from https://cakeml.org/. Citation: pg. 211.

Oskar Abrahamsson, Johannes Åman Pohjola, Anthony Fox, Alejandro Gómez-Londoño, Hrutvik Kanabar, Ramana Kumar, Andreas Lööw, Magnus Myreen, Michael Norrish, Scott Owens, Thomas Sewell, Hira Syeda, Yong Kiam Tan, Timotej Tomandl, and other contributors. CakeML: A verified implementation of ML. August 5, 2020b. URL https://cakeml.org/. Date is of last access. Citation: pg. 214.

Stephen Adams. Functional pearls: Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, 1993. DOI: 10.1017/S0956796800000885. URL https://doi.org/10.1017/S0956796800000885. Citation: pg. 189.

Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A verified compiler for Coq. In *Proceedings of the Third International Workshop on Coq for Programming Languages (CoqPL '17)*. ACM, 2017. URL https://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf. Citations: pp. 10 and 183.

Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20160331, September 4, 2017. DOI: 10.1098/rsta.2016.0331. URL https://royalsocietypublishing.org/doi/10.1098/rsta.2016.0331. Citations: pp. 9, 9, 124, 129, 217, 218, and 218.

Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated*

Reasoning (in 2 volumes), pages 1149–1238. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9. DOI: 10.1016/b978-044450813-3/50020-5. URL https://doi.org/10.1016/b978-044450813-3/50020-5. Citation: pg. 1.

Tobias Baum, Hendrik Leßmann, and Kurt Schneider. The choice of code review process: A survey on the state of the practice. In Michael Felderer, Daniel Méndez Fernández, Burak Turhan, Marcos Kalinowski, Federica Sarro, and Dietmar Winkler, editors, *Product-Focused Software Process Improvement - 18th International Conference, PROFES 2017, Innsbruck, Austria, November 29 - December 1, 2017, Proceedings*, volume 10611 of *Lecture Notes in Computer Science*, pages 111–127. Springer, 2017. ISBN 978-3-319-69925-7. DOI: 10.1007/978-3-319-69926-4_9. URL https://doi.org/10.1007/978-3-319-69926-4_9. Citation: pg. 1.

Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010. ISBN 978-3-642-14051-8. DOI: 10.1007/978-3-642-14052-5_11. URL https://doi.org/10.1007/978-3-642-14052-5_11. Citation: pg. 84.

Joachim Breitner. `successors`: An applicative functor to manage successors (version 0.1.0.1). December 31, 2017. URL http://hackage.haskell.org/package/successors-0.1.0.1. Citations: pp. 51, 51, 52, 53, and 53.

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. Ready, `Set`, verify! Applying `hs-to-coq` to real-world Haskell code (experience report). *Proceedings of the ACM on Programming Languages*, 2(ICFP):89:1–89:16, 2018. DOI: 10.1145/3236784. URL https://doi.org/10.1145/3236784. Citations: pp. 11, 21, 34, 48, 71, 71, 92, 95, 175, and 176.

Lukas Bulwahn. The new Quickcheck for Isabelle: Random, exhaustive and symbolic testing under one roof. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2012. ISBN 978-3-642-35307-9. DOI: 10.1007/978-3-642-35308-6_10. URL https://doi.org/10.1007/978-3-642-35308-6_10. Citations: pp. 84 and 84.

Arthur Charguéraud. Program verification through characteristic formulae. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 321–332. ACM, 2010. ISBN 978-1-60558-794-3. DOI: 10.1145/1863543.1863590. URL https://doi.org/10.1145/1863543.1863590. Citations: pp. 198, 198, 198, and 199.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000. ISBN 1-58113-202-6. DOI: 10.1145/351240.351266. URL https://doi.org/10.

1145/351240.351266. Citations: pp. 1, 75, 75, 84, 84, 85, and 85.

Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: an approach to high-assurance microkernel development. In Andres Löh, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2006, Portland, Oregon, USA, September 17, 2006*, pages 60–71. ACM, 2006. ISBN 1-59593-489-8. DOI: 10.1145/1159842.1159850. URL https://doi.org/10.1145/1159842.1159850. Citation: pg. 190.

Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying Haskell programs by combining testing, model checking and interactive theorem proving. *Information & Software Technology*, 46(15):1011–1025, 2004. DOI: 10.1016/j.infsof.2004.07.002. URL https://doi.org/10.1016/j.infsof.2004.07.002. Citations: pp. 195, 195, 195, and 195.

Kevin Elphinstone, Gerwin Klein, and Rafal Kolanski. Formalising a high-performance microkernel. Technical report, Verified Software: Theories, Tools, And Experiments (VSTTE) 2006 Workshop Proceedings, Microsoft Research (MSR-TR-2006-117), August 2006. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.61.6433. Citations: pp. 190, 191, and 192.

Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In Galen C. Hunt, editor, *Proceedings of HotOS'07: 11th Workshop on Hot Topics in Operating Systems, May 7-9, 2005, San Diego, California, USA*. USENIX Association, 2007. URL http://www.usenix.org/events/hotos07/tech/full_papers/elphinstone/elphinstone.pdf. Citation: pg. 190.

Andy Gill, Simon Marlow, and other contributors. Happy: The parser generator for Haskell (version 1.18.5). June 17, 2010. URL https://www.haskell.org/happy/. Citation: pg. 22.

Georges Gonthier. Formal proof—the four-color theorem. *Notices of the AMS*, 55(11): 1382–1393, 2008. Citation: pg. 2.

Florian Haftmann. From higher-order logic to Haskell: there and back again. In John P. Gallagher and Janis Voigtländer, editors, *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, January 18-19, 2010*, pages 155–158. ACM, 2010. DOI: 10.1145/1706356.1706385. URL https://doi.org/10.1145/1706356.1706385. Citations: pp. 188, 189, and 189.

Thomas Hallgren, James Hook, Mark P Jones, and Richard B Kieburtz. An overview of the Programatica toolset. In *Presented at the Fourth Annual High Confidence Software and Systems Conference (HCSS '04)*, 2004. URL http://ogi.altocumulus.org/~hallgren/Programatica/HCSS04/. Citations: pp. 192, 193, 193, 193, and 195.

William L. Harrison and Richard B. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(5):837–891, 2005. DOI: 10.1017/S0956796805005666. URL https://doi.org/10.1017/S0956796805005666. Citations: pp. 195 and 195.

HaskellWiki contributors. Functor-applicative-monad proposal - haskellwiki. October 24, 2015. URL https://wiki.haskell.org/Functor-Applicative-Monad_Proposal. Citation: pg. 49.

Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, 2007. DOI: 10.1145/1278901.1278904. URL https://doi.org/10.1145/1278901.1278904. Citation: pg. 190.

Hugo Herbelin <Hugo.Herbelin@inria.fr>. Re: [coq-club] termination checking with nested recursion. September 25, 2010. URL https://sympa.inria.fr/sympa/arc/coq-club/2010-09/msg00111.html. Originally an email to the "coq-club" mailing list (https://sympa.inria.fr/sympa/info/coq-club). Citation: pg. 107.

Brian Huffman, John Matthews, and Peter White. Axiomatic constructor classes in Isabelle/HOLCF. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2005. ISBN 3-540-28372-2. DOI: 10.1007/11541868_10. URL https://doi.org/10.1007/11541868_10. Citation: pg. 191.

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 193–206. ACM, 2013. ISBN 978-1-4503-1832-7. DOI: 10.1145/2429069.2429093. URL https://doi.org/10.1145/2429069.2429093. Citation: pg. 33.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, 2016. ISBN 978-1-31-662622-1. DOI: 10.1017/CBO9780511813672. Citations: pp. 55, 57, 209, and 209.

Pepe Iborra. `FiniteMap`: A finite map implementation, derived from the paper: Efficient sets: a balancing act, S. Adams, Journal of functional programming 3(4) Oct 1993, pp553-562 (version 0.1). March 6, 2007. URL http://hackage.haskell.org/package/FiniteMap-0.1. Citation: pg. 189.

Spencer Janssen, Don Stewart, Adam Vogt, Brent Yorgey, Daniel Wagner, David Roundy, Daniel Schoepe, Eric Mertens, Nicolas Pouillard, Roman Cheplyaka, Gwern Branwen, Lukas Mai, Braden Shepherdson, and Devin Mullins. xmonad | the tiling window manager that rocks. March 6, 2021. URL https://xmonad.org/. Date is of last access. Citation: pg. 2.

Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009. DOI: 10.1145/1592434.1592438. URL https://doi.org/10.1145/1592434.1592438. Citation: pg. 1.

Ranjit Jhala (`ranjitjhala`), Niki Vazou (`nikivazou`), Eric L. Seidel (`gridaphobe`), and other contributors. Liquid types for Haskell (GitHub repository). August 5, 2020. URL https://github.com/ucsd-progsys/liquidhaskell. Date is of last access (commit `1cab7e3758`). Citations: pp. 201 and 202.

Richard B. Kieburtz. P-logic: property verification for Haskell programs. Older version available from http://programatica.cs.pdx.edu/papers.html, August 14, 2002. URL http://https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.3152&rep=rep1&type=pdf. Citations: pp. 193 and 195.

Gerwin Klein. Operating system verification—an overview. *Sadhana*, 34(1):27–69, 2009. DOI: 10.1007/s12046-009-0002-4. URL https://link.springer.com/article/

10.1007/s12046-009-0002-4. Citation: pg. 190.

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010. DOI: 10.1145/1743546.1743574. URL https://doi.org/10.1145/1743546.1743574. Citation: pg. 190.

Edward A. Kmett. `lens`: Lenses, folds and traversals (version 4.16.1). March 23, 2018. URL http://hackage.haskell.org/package/lens-4.16.1. Citation: pg. 22.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014. ISBN 978-1-4503-2544-8. DOI: 10.1145/2535838.2535841. URL https://doi.org/10.1145/2535838.2535841. Citations: pp. 125, 211, 211, 211, 212, 212, 212, and 213.

Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*, volume 4 of *Software Foundations*. Electronic textbook, version 1.0 edition, August 9, 2018. URL https://softwarefoundations.cis.upenn.edu/qc-current/index.html. Citations: pp. 84 and 84.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. DOI: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814. Citations: pp. 2, 125, and 211.

Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2002. ISBN 3-540-14031-X. DOI: 10.1007/3-540-39185-1_12. URL https://doi.org/10.1007/3-540-39185-1_12. Citations: pp. 183, 183, and 183.

Nancy G. Leveson and Clark S. Turner. Investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993. DOI: 10.1109/MC.1993.274940. URL https://doi.org/10.1109/MC.1993.274940. Citation: pg. 1.

Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. Verifying replicated data types with typeclass refinements in Liquid Haskell. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):216:1–216:30, 2020. DOI: 10.1145/3428284. URL https://doi.org/10.1145/3428284. Citations: pp. 210, 210, 210, 210, 210, 210, and 210.

John MacFarlane. Pandoc user's guide. Available from https://pandoc.org/, January 21, 2021. URL https://pandoc.org/MANUAL.pdf. Citation: pg. 2.

Simon Marlow and Simon Peyton-Jones. The Glasgow Haskell Compiler. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications*, volume II. Available online under the Creative Commons Attribution 3.0 Unported license, March 30, 2012. URL http://www.aosabook.org/en/ghc.html. Citations: pp. 2 and 87.

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. Compiling without continuations. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 482–494. ACM, 2017. ISBN 978-1-4503-4988-8. DOI: 10.1145/3062341.3062380. URL https://doi.org/10.1145/3062341.3062380. Citation: pg. 127.

Donald R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968. DOI: 10.1145/321479.321481. URL https://doi.org/10.1145/321479.321481. Citation: pg. 71.

Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 115–126. ACM, 2012. ISBN 978-1-4503-1054-3. DOI: 10.1145/2364527.2364545. URL https://doi.org/10.1145/2364527.2364545. Citations: pp. 211 and 212.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999. ISBN 978-0-521-66350-2. Citations: pp. 199 and 199.

Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998. URL https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.5452. Citation: pg. 71.

Will Partain (`partain`). GHC commit `6c381e873e`: "simonpj/sansom/partain/dnt 1.3 compiler stuff through 96/03/18". March 19, 1996. URL https://gitlab.haskell.org/ghc/ghc/-/commit/6c381e873e222417d9a67aeec77b9555eca7b7a8. Citation: pg. 62.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2001, Firenze, Italy, September 2, 2001*. ACM SIGPLAN, 2001. URL https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/. Citation: pg. 99.

Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992. DOI: 10.1017/S0956796800000319. URL https://doi.org/10.1017/S0956796800000319. Citation: pg. 87.

Simon L. Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4&5):393–433, 2002. DOI: 10.1017/S0956796802004331. URL https://doi.org/10.1017/S0956796802004331. Citation: pg. 88.

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. ACM, 2008. ISBN 978-1-59593-860-2. DOI: 10.1145/1375581.1375602. URL https://doi.org/10.1145/1375581.1375602. Citation: pg. 199.

Per Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006. DOI: 10.1109/MS.2006.91. URL https://doi.org/10.1109/MS.2006.91. Citation: pg. 1.

Tom Smalley (`tomsmalley`). Data.IntMap.restrictKeys and Data.IntMap.withoutKeys not working (issue #392, haskell/containers). February 6, 2017. URL https://github.com/haskell/containers/issues/392. Citation: pg. 84.

Matthieu Sozeau. Subset coercions in Coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2006. ISBN 978-3-540-74463-4. DOI: 10.1007/978-3-540-74464-1_16. URL https://doi.org/10.1007/978-3-540-74464-1_16. Citations: pp. 6, 175, and 179.

Matthieu Sozeau and Cyprien Mangin. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.*, 3 (ICFP):86:1–86:29, 2019. DOI: 10.1145/3341690. URL https://doi.org/10.1145/3341690. Citations: pp. 38 and 223.

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is reasonable Coq. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 14–27. ACM, 2018. ISBN 978-1-4503-5586-5. DOI: 10.1145/3167092. URL https://doi.org/10.1145/3167092. Citations: pp. 11, 12, 21, 33, 45, 45, and 51.

Antal Spector-Zabusky, Joachim Breitner, Yao Li, and Stephanie Weirich. Embracing a mechanized formalization gap: Pragmatic software system verification (extended version). Unpublished work, October 25, 2019. URL https://arxiv.org/abs/1910.11724. Citations: pp. 11, 21, and 33.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In François Pottier and George C. Necula, editors, *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, pages 53–66. ACM, 2007. ISBN 1-59593-393-X. DOI: 10.1145/1190315.1190324. URL https://doi.org/10.1145/1190315.1190324. Citation: pg. 88.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 60–73. ACM, 2016. ISBN 978-1-4503-4219-3. DOI: 10.1145/2951913.2951924. URL https://doi.org/10.1145/2951913.2951924. Citations: pp. 125, 211, 211, 211, 211, 212, 212, 214, and 214.

The Coq Development Team. The Coq standard library, version 8.11.0. January 2020a. URL https://coq.github.io/doc/v8.11/stdlib/. Citation: pg. 109.

The Coq Development Team. The Coq proof assistant, version 8.11.0. January 30, 2020b. URL https://coq.inria.fr/refman/index.html. Citations: pp. 176, 183, 186, and 200.

The Core Libraries Committee. `base`: Basic libraries (version 4.11.1.0). April 21, 2018. URL http://hackage.haskell.org/package/base-4.11.1.0. Citations: pp. 46, 46, 50, 51, and 115.

The Programatica Team. Programatica tools for certifiable, auditable development of high-assurance systems in Haskell. Available from http://programatica.cs.pdx.edu/papers.html, 2003. URL http://programatica.cs.pdx.edu/P/ProgramaticaAssurance.pdf. Citation: pg. 193.

Simon J. Thompson. Formulating Haskell. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, 6-8 July 1992*, Workshops in Computing, pages 258–268. Springer, 1992. ISBN 3-540-19820-2. DOI: 10.1007/978-1-4471-3215-8_23. URL https://doi.org/10.1007/978-1-4471-3215-8_23. Citations: pp. 198 and 198.

Simon J. Thompson. A logic for Miranda, revisited. *Formal Aspects of Computing*, 7 (4):412–429, 1995. DOI: 10.1007/BF01211216. URL https://doi.org/10.1007/BF01211216. Citations: pp. 198, 198, and 198.

Travis CI, GmbH. Travis ci - test and deploy with confidence. 2020. URL https://travis-ci.com/. Date is of last access. Citation: pg. 25.

Alan Matheson Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, January 1, 1937. ISSN 0024-6115. DOI: 10.1112/plms/s2-42.1.230. URL https://doi.org/10.1112/plms/s2-42.1.230. Citations: pp. 201 and 214.

Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. Abstract refinement types. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2013. ISBN 978-3-642-37035-9. DOI: 10.1007/978-3-642-37036-6_13. URL https://doi.org/10.1007/978-3-642-37036-6_13. Citations: pp. 209, 209, and 209.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for Haskell. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (ICFP), Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014. ISBN 978-1-4503-2873-9. DOI: 10.1145/2628136.2628161. URL https://doi.org/10.1145/2628136.2628161. Citation: pg. 199.

Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. A tale of two provers: verifying monoidal string matching in Liquid Haskell and Coq. In Iavor S. Diatchki, editor, *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 63–74. ACM, 2017. ISBN 978-1-4503-5182-9. DOI: 10.1145/3122955.3122963. URL https://doi.org/10.1145/3122955.3122963. Citations: pp. 199, 200, 200, 200, and 200.

Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Functional pearl: Theorem proving for all: Equational reasoning in Liquid Haskell. In Nicolas Wu, editor, *Proceedings of the 11th ACM SIGPLAN International*

Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018, pages 132–144. ACM, 2018a. DOI: 10.1145/3242744.3242756. URL https://doi.org/10.1145/3242744.3242756. Citations: pp. 209 and 209.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2 (POPL):53:1–53:31, 2018b. DOI: 10.1145/3158141. URL https://doi.org/10.1145/3158141. Citations: pp. 199 and 203.

Freek Wiedijk, editor. *The Seventeen Provers of the World*. Springer-Verlag Berlin Heidelberg, 2006. ISBN 978-3-540-32888-9. DOI: 10.1007/11542384. URL https://www.springer.com/gp/book/9783540307044. Citation: pg. 1.

Kazu Yamamoto. `llrbtree`: Purely functional sets and heaps (version 0.1.1). January 31, 2012. URL http://hackage.haskell.org/package/llrbtree-0.1.1. Citation: pg. 209.