MECHANIZED REASONING ABOUT "HOW" USING FUNCTIONAL PROGRAMS AND EMBEDDINGS

Yao Li

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

2022

Supervisor of Dissertation

Stephanie Weirich, ENIAC President's Distinguished Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Steve Zdancewic, Schlein Family President's Distinguished Professor and Associate Chair
Benjamin C. Pierce, Henry Salvatori Professor of Computer and Information Science
Andre Scedrov, Professor of Mathematics, Professor of Computer and Information Science
Nikhil Swamy, Senior Principal Researcher, Microsoft Research

MECHANIZED REASONING ABOUT “HOW” USING FUNCTIONAL PROGRAMS AND EMBEDDINGS

ACKNOWLEDGEMENT

I feel extremely grateful to my advisor Stephanie Weirich, who has given me a lot of guidance on everything about research and has been a huge support for me during the tough times. Thank you, Stephanie, for teaching me everything I know about research, for supporting me on exploring research topics that I am passionate about, and for always believing in me.

I want to thank my mentors during the DeepSpec project, Benjamin C. Pierce and Steve Zdancewic. I have learned a lot from them and, as you will see in this dissertation, my collaboration with them has contributed significantly to my vision that guides this dissertation. In particular, I have worked closely with Benjamin during DeepSpec and he has given me a lot of operational advice that is well suited for me. I find myself keep recalling his advice throughout the six years of my Ph.D. journey and I am sure this will continue as I move on to a new phase in my research career.

I also want to thank the members of my dissertation committee, including Benjamin C. Pierce, Andre Scedrov, Nikhil Swamy, and Steve Zdancewic, who have given me great feedback. I would also like to thank Val Tannen, who has given me many useful suggestions during my proposal.

I owe my gratitude to all other amazing collaborators that I had the pleasure to work with during my Ph.D., including Andrew Appel, Lennart Beringer, Jay Bosamiya, Joachim Breitner, Qinxiang Cao, Koen Claessen, Joshua Cohen, Joseph W. Cutler, Sydney Gibson, Chris Hawblitzel, Wolf Honoré, Nicolas Koh, Yishuai Li, William Mansky, Bryan Parno, Christine Rizkallah, Antal Spector-Zabusky, John Wiegley, Li-yao Xia, and Hengchu Zhang. Collaborating with them has been both enlightening and fun.

Pursuing a Ph.D. in another country has not been easy because I had to leave almost my entire supportive network on the other side of earth. However, I am extremely lucky to have met many amazing friends here. I owe my thanks to Calvin Beck, Joachim Breitner, Haoxian Chen, Leshang Chen, Pritam Choudhury, Jiajing Gao, Hangfeng He, Paul He, Jiani Huang, Kishor Jothimurugan, Konstantinos Kallas, Nicolas Koh, Dong-Ho Lee, Omar Navarro Leija, Yishuai Li, Hui Lyu, Solomon Maina, Filip Niksic, Christine Rizkallah, Clara Schneidewind, Lei Shi, Xujie Si, Antal Spector-Zabusky, Caleb Stanford, Aalok Thakkar, Antoine Voizard, Yinjun Wu, Li-yao Xia, Irene Yoon, Yannick Zakowski, Hengchu Zhang, Qicheng Zhang, Qizhen Zhang, Richard Zhang, Teng Zhang, Yi Zhang, Xin Zhang, Xiaozhen Zhu, and many others. I'd also like to extend my gratitude to all members of PLClub, many students and faculty members in the Department of Computer and Information Science at Penn, who have made me feel that I belong here as a member of a big happy family.

Besides being at Penn, I had a wonderful summer at Microsoft Research as an intern in 2018, where I had a lot of fun working with Chris Hawblitzel, Guido Martínez, Zoe Paraskevopoulou, Jonathan Protzenko, Tahina Ramananandro, Nikhil Swamy, and many other researchers there.

I have not been able to meet with my family often during my Ph.D., especially since 2020, when the COVID-19 pandemic broke out. They have, nevertheless, been very supportive to me and for my choice to pursue a career as an academic.

I was blessed to meet with my girlfriend, Bingzhe Liu, during my Ph.D. As someone who understands my dream and someone who resonates with me on almost every level, she has helped me countless times when I was confused about my path. Her support was crucial for me to get through the toughest times. Thank you, Bingzhe, for everything. I love you.

Bingzhe and I adopted a cat, Mango, in 2021, who has brought a lot of joy to our place since then. I owe my special thanks to Mango as well, for she has literally been by my side almost the entire time when I was working on this dissertation, asking for belly rubs whenever she found me struggling with a sentence.

ABSTRACT

MECHANIZED REASONING ABOUT "HOW" USING FUNCTIONAL PROGRAMS AND EMBEDDINGS

Yao Li

Stephanie Weirich

Embedding describes the process of encoding a program's syntax and/or semantics in another language—typically a theorem prover in the context of mechanized reasoning. Among different embedding styles, deep embeddings are generally preferred as they enable the most faithful modeling of the original language. However, deep embeddings are also the most complex, and working with them requires additional effort. In light of that, this dissertation aims to draw more attention to alternative styles, namely shallow and mixed embeddings, by studying their use in mechanized reasoning about programs' properties that are related to "how". More specifically, I present a simple shallow embedding for reasoning about computation costs of lazy programs, and a class of mixed embeddings that are useful for reasoning about properties of general computation patterns in effectful programs. I show the usefulness of these embedding styles with examples based on real-world applications.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

Suppose that you want to mechanically reason about a program. The program might be written in C, Java, Haskell, Verilog, or Makefile, *etc.* Your first step would be to embed the language's syntax and/or semantics in a language that is amenable to mechanized reasoning, such as Coq (Coq development team, 2022), Isabelle (Nipkow et al., 2002), HOL (Gordon, 2000), or F$\star$ (Swamy et al., 2013a). This step is known as *embeddings* (Boulton et al., 1992).

We commonly categorize all embedding styles into two types following the categorization of Boulton et al. (1992). They are: *shallow* embeddings, which directly use "equivalent" terms of the embedding language to denote the original language, and *deep* embeddings, which represent the original language using an abstract syntax tree (AST). Besides these two styles, more recent works (Chlipala, 2021; Pfenning and Elliott, 1988; Prinz et al., 2022; Washburn and Weirich, 2008) point out that there are also many embedding styles that mix shallow and deep embeddings. I call these styles *mixed* embeddings.

Among these embedding styles, deep embeddings are commonly used in mechanized reasoning because they allow us to prove both syntactic and semantic properties, and because we do not need to rely on the semantics of the embedding language to coincide with the original language. However, defining and reasoning about the AST requires nontrivial effort for most languages and the terms in deep embeddings are hard to work with in mechanized reasoning. These limitations make deep embeddings a daunting choice.

In this dissertation, I aim to draw more attention to alternative embedding styles, namely shallow embeddings and mixed embeddings. The dissertation is *not* about preferring other embedding styles over deep embeddings though. Every embedding style has its advantages and disadvantages, so choosing an embedding style is mostly a design decision tailored to the requirements of each individual application. However, I believe that shallow embeddings and mixed embeddings are more useful than most people anticipate. The belief is built on my experience with two projects that use shallow embeddings and mixed embeddings. These two projects are also direct inspirations to the primary works presented in this dissertation.

This dissertation shows the usefulness of shallow and mixed embeddings by studying their use in mechanized reasoning about programs' properties related to "how". I present two works: (1) a simple shallow embedding for reasoning about computation cost of lazy programs; and (2) a class of mixed embeddings for reasoning about general properties about computation patterns in effectful programs. I show the usefulness of these embedding styles with examples based on real-world applications.

1.1. Mechanized Reasoning

Mechanized reasoning allows us to verify that a program follows its specification. To mechanically reason about an existing program written in another language, we first embed a mathematical model of our program's syntax and/or semantics in a *theorem prover* or a *proof assistant* such as Coq, Isabelle, HOL, or F⋆. A theorem prover typically contains an expressive language so that we can mathematically describe the program's correct behavior—the description is also known as a *formal specification*—and for writing mechanized proofs that would be examined by an automatic proof checker. A distinguishing appeal of the method is that, once we establish a correctness proof of a program, we have high confidence that the program is correct with respect to the specification such that relevant bugs are *absent*.

All works presented in this dissertation is based on an interactive theorem prover called Coq (Coq development team, 2022). Coq is equipped with a dependently-typed purely functional programming language that allows us to write rich specifications that describe a program's complex behaviors in detail. It also contains a tactic language for writing proof scripts. In addition, there is a rich ecosystem built around Coq (Appel, 2022; Ringer et al., 2019a) that includes libraries, plugins, frameworks for mechanized reasoning, *etc.* For these reasons, Coq has been used in many works on mechanized reasoning, including the verified compiler CompCert (Leroy, 2009), the verified operating system CertiKOS (Gu et al., 2019), and the verified file system FSCQ (Chen et al., 2015), *etc.* These are also important reasons that shallow and mixed embeddings presented in this dissertation would work nicely—as these embedding styles enable reusing Coq's language features and its rich ecosystem.

In this dissertation, I assume that readers already have basic familiarity with Coq. Interested readers who would like to learn about the basics of Coq should refer to the *Software Foundations* textbook series (Pierce et al., 2021a,b) or other Coq textbooks such as Bertot and Castéran (2004); Chlipala (2019, 2022).

1.2. Functional Programs

Programming is a task that demands us to simultaneously focus on the aspects of "what" and "how". It is important that, given an input, a program returns us the correct answer. But that is rarely enough. We also expect the program to return the correct answer in the right way: we want the answer to be given within a reasonable time; we want the program to not delete our valuable files or take up all the memory; and if the program communicates with another program, we want it to abide to an established protocol.

Functional programming, however, embraces a focus on "what": what is a value `x`, what is a function `f`, and what `f` applied to `x` evaluates to. We say that `1 + 0` and `1` are "equal" because *what* they evaluate to are equal—even though *how* they evaluate to the same result are different. This focus on "what" is a useful bias. For example, a pure functional program is *referentially transparent*, which means that a value can be evaluated at any time. This makes it possible to use evaluation strategies such as lazy evaluation (Henderson and Morris, 1976) to write compositional code (Hughes, 1989). Even for programs with effects, this is often true for the pure part. Furthermore, the bias also makes reasoning easier, as it enables some simple but powerful techniques like equational reasoning (Gibbons and Hinze, 2011; Gonzalez, 2013; Vazou et al., 2018a; Wadler, 1987).

A challenge for using functional programming languages like Coq to reason about other programs' properties that are related to "how" is that we need a way to make the evaluation strategy explicit. Fortunately, the solution is known: we can embed those original programs using monads (Moggi, 1991; Wadler, 1992) and other classes of functors (McBride and Paterson, 2008; Mokhov et al., 2019) that reify the evaluation strategies. We will cover this technique in more detail in Chapter 2.

In this dissertation, I assume that readers are already familiar with basics of functional programming, including algebraic data types (Burstall et al., 1980), type classes (Morris, 2013; Wadler and Blott, 1989), monads, *etc.*

1.3. Contributions

The contributions of this dissertation are as follow:

- I, along with my collaborators, build a simple shallow embedding based on a new model of lazy evaluation (Hackett and Hutton, 2019) for reasoning about computation costs of lazy programs (Li, Xia, and Weirich, 2021a). The embedding is based on a monad called the clairvoyance monad, whose key definitions can be defined using around 20 lines of code in Coq. We define the embedding rules for embedding a language with structural recursion in Coq using the clairvoyance monad. We also develop a dual reasoning framework for analyzing the computation cost of an embedded program in a local and modular way.

- I, along with my advisor, identify a class of mixed embeddings called Tlön embeddings for modeling effectful programs. Tlön embeddings are based on a class of data structures that capture a variety of general computation patterns called program adverbs (Li and Weirich, 2022a). Program adverbs are themselves composable, allowing them to be used to specify the semantics of languages with multiple computation patterns. Tlön embeddings allow flexibility in computational modeling of effects, while retaining more information about the program's syntactic structure.

- Both works mentioned above are accompanied by our Coq development, which have been made publicly available as artifacts (Li and Weirich, 2022b; Li, Xia, and Weirich, 2021b). These artifacts have been reviewed by artifact evaluation committees.

- Besides these works, I also contributed to a number of other works (Breitner, Spector-Zabusky, Li, Rizkallah, Wiegley, Cohen, and Weirich, 2021; Koh, Li, Li, Xia, Beringer, Honoré, Mansky, Pierce, and Zdancewic, 2019; Spector-Zabusky, Breitner, Li, and Weirich, 2019; Zhang, Honoré, Koh, Li, Li, Xia, Beringer, Mansky, Pierce, and Zdancewic, 2021) that are direct inspirations that lead me to the primary contributions presented in this dissertation.

1.4. Outline

The rest of the dissertation is organized as follows:

- I discuss the concept of embeddings, compare different embedding styles, and illustrate the use of monads as a way of reifying evaluation strategies in an embedding in Chapter 2.

- I summarize two projects that I have contributed to in Chapter 3. I also discuss how they formulated my views on embeddings.

- I present our work on analyzing the computation cost of lazy programs in Chapter 4.

- I present our work on mixed embeddings for effectful programs in Chapter 5.

- I list all the primary related work in Chapter 6 and talk about potential future work in Chapter 7. I wrap up the dissertation in Chapter 8.

CHAPTER 2

Embeddings

My focus in this dissertation is in reasoning about "how" in *existing* programs. Here is the first challenge: existing programs that we would like to reason about are typically *not* written in languages amenable to mechanized reasoning. These languages include most mainstream programming languages, such as C, Java, Haskell, Verilog, or Makefile *etc.* In order to reason about programs written in those languages, we need to embed the syntax and/or semantics of that language in a theorem prover such as Coq.

In this dissertation, I use the *verb* "embedding" to describe the process of encoding a language's syntax and/or semantics using another language; I use the *noun* "an embedding" or "embeddings" to describe everything involved in the embedding process.[1]

I show an overview of a typical embedding in Fig. 2.1. A typical embedding includes the following components: An **original program**, which is the program we would like to embed; An **original language** (also called the object language), which is the programming language that the original program is written in; An **embedding language** (also called the meta language or the host language), which is the programming language we use to embed the original program; An **embedding domain**, which is the data type(s) we use to embed the original program in the embedding language; **embedding rules**, which are "the recipe" of how to embed an original language in an embedding domain; Finally, an **embedded program**, which is the result of **embedding** the original program following embedding rules. We will be using these terms for describing embeddings throughout the dissertation.

Boulton et al. (1992) categorizes embeddings into two types based on differences in their embedding domains. They are: (1) *shallow* embeddings, whose embedding domain is just

[1]Boulton et al. (1992) use the term "semantic embedding" instead of "embedding". I intentionally choose not to follow the term of Boulton et al. because a semantic embedding might give a reader the wrong impression that an embedding is all about semantics.

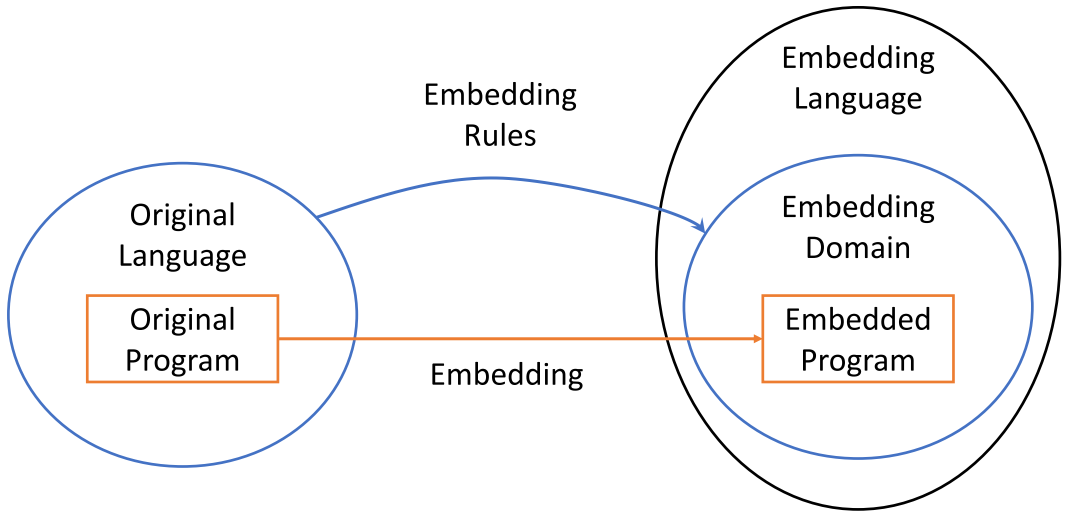

Figure 2.1: An overview of an embedding.

terms of the embedding language, and (2) *deep* embeddings, whose embedding domain is an abstract syntax tree (AST) with an interpreter that defines the semantics.

There are, however, not just two embedding styles. Later, many works (Chlipala, 2021; Pfenning and Elliott, 1988; Prinz et al., 2022; Washburn and Weirich, 2008) point out that there are also many embedding styles between shallow and deep embeddings. In this dissertation, I call all these embedding styles *mixed embeddings* following the terminology of Chlipala (2021).

To illustrate different embedding styles, let's consider a simple language $\mathcal{N}$, which encodes an arithmetic on natural numbers. I show the syntax of $\mathcal{N}$ in Fig. 2.2. Semantically, we want the arithmetic operators to have their usual semantics. For any $m, n \in \mathbb{N}$ such that $m < n$, $m - n$ is defined as 0. We use the notation $[\![\cdot]\!]$ to represent embedding rules.

2.1. Shallow Embeddings

In our first example, we choose Coq's built-in type `nat` for representing natural numbers as the embedding domain. We show our embedding rules in Fig. 2.3. Infix operators `+`, `-` are

$$
\begin{array}{lrl}
\textit{literals} & n & \in \mathbb{N} \\
\textit{terms} & t, u & ::= \; n \mid t + u \mid t - u
\end{array}
$$

Figure 2.2: The syntax of $\mathcal{N}$.

Original Language: $\mathcal{N}$

Embedding Domain:

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

Embedding Language: Coq

Embedding Rules:

$$
\begin{aligned}
[\![n]\!] &= \texttt{n} \\
[\![t + u]\!] &= [\![t]\!]\texttt{+}[\![u]\!] \\
[\![t - u]\!] &= [\![t]\!]\texttt{-}[\![u]\!]
\end{aligned}
$$

Figure 2.3: A shallow embedding from $\mathcal{N}$ to Coq's `nat` type.

notations for Coq's function that computes addition and subtraction of natural numbers. For any `m n : nat` such that `m < n`, `m - n` is defined to return `0`.

Since we use Coq's `nat` type to directly encode $\mathcal{N}$'s semantics, we also call the `nat` type the *semantic domain* of this embedding. A shallow embedding's semantic domain coincides with its embedding domain.

An advantage of shallow embeddings is that, since their embedding domains are terms of the embedding language, we can reuse theories and other language mechanisms available in the embedding language. In our example, we can directly use existing theorems (*e.g.,* various arithmetic laws such as commutativity for $+$) in Coq's library for `nat`; we can also use tactics provided by Coq and its libraries, such as `lia`, a decision procedure for linear integer arithmetic (Besson, 2006).

However, because we directly encode an original language's semantics in a shallow embedding, we do not have a structure for representing the original language's syntax. Indeed, the `nat` type used in our example can only distinguish different natural numbers—$[\![1 + 2]\!]$ and $[\![4 - 1]\!]$ are just the same thing in our embedding domain. This makes shallow embeddings rather limiting in scenarios where we would like to state properties related to an original program's syntactic structure, *e.g.,* whether an expression contains a subtraction.

Original Language: $\mathcal{N}$

Embedding Language: Coq

Embedding Domain:

```
Inductive term : Set :=
| Lit (n : nat)
| Plus  (t u : term)
| Minus (t u : term).
```

Embedding Rules:

$$[\![n]\!] = \texttt{Lit n}$$
$$[\![t+u]\!] = \texttt{Plus}\ [\![t]\!]\ [\![u]\!]$$
$$[\![t-u]\!] = \texttt{Minus}\ [\![t]\!]\ [\![u]\!]$$

Semantic Domain:

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

Interpretation:

```
Fixpoint interp (t : term) : nat :=
  match t with
  | Lit b => b
  | Plus  t u => interp t + interp u
  | Minus t u => interp t - interp u
  end.
```

Figure 2.4: A deep embedding from $\mathcal{N}$ to our `term` type in Coq.

Another challenge for a shallow embedding is that, more often than not, there are discrepancies between the original language and the embedding language that makes a direct embedding not possible: there might be features in the original language that are not supported by the embedding language, or the type checker in the original language works differently from that of the embedding language, *etc.*

2.2. Deep Embeddings

Alternatively, we can make a deep embedding by defining a `term` data type in Coq that represents the AST of $\mathcal{N}$. We show our deep embedding of $\mathcal{N}$ in Fig. 2.4.

The `term` data type shown in Fig. 2.4 has a case for every syntax construct of $\mathcal{N}$ (Fig. 2.2) and the embedding rules are a straightforward translation from one syntax to another syntax. We still reuse Coq's `nat` type for literals in $\mathcal{N}$ here. We can also choose to define our own data type but that would just be isomorphic to `nat`. Since the `term` data type only encodes the syntax, we need an additional semantic domain and an interpretation from `term` to the semantic domain if we would like to reason about $\mathcal{N}$'s semantics. Figure 2.4 also shows such an interpretation (function `interp`) if we choose `nat` as the semantic domain.

Compared with our previous shallow embedding, the extra `term` data type in our deep embedding makes it possible to state and reason about the syntactic structure of an original program. For example, we can define the following function that checks if a program does *not* contain a subtraction:

```
Fixpoint no_minus (t : term) : bool :=
  match t with
  | Lit _ => true
  | Plus  t u => no_minus t && no_minus u
  | Minus t u => false
  end.
```

The infix operator `&&` is the Boolean "and" operation defined on Coq's `bool` type.

Another benefit of a deep embedding is that when we would like change the semantic domain in our embedding, we can do that by changing the interpretation without changing the embedding rules. Therefore, a deep embedding is more *modular* than a shallow embedding.

On the other hand, one main issue with a deep embedding is the extra effort required for defining and working with the AST. Our example here is intentionally simple, but it can quickly get complicated when we would like to add functions or let bindings, which require us to represent and reason about variable bindings and capture-avoiding substitutions (Aydemir et al., 2005), or when other complicated features are added. Furthermore, since an AST is—unlike `nat`—not a builtin type in Coq, we would not have existing theorems or tactics that can be directly used. Despite many existing tools (Aydemir et al., 2008; Schäfer et al., 2015; Sewell et al., 2010) that help with this process, the task of working with a deep embedding remains nontrivial.

2.3. Mixed Embeddings

We do not have to commit to either a shallow or a deep embedding, as we can make certain parts of our embedding shallow while keeping other parts deep. There are many different ways of achieving this mixture (Chlipala, 2008, 2021; Dylus et al., 2019; Harper et al., 1987;

variables	x, y, z		
literals	n		$\in \mathbb{N}$
terms	t, u	$::=$	$x \mid n \mid t + u \mid t - u \mid \texttt{let}\ x = t\ \texttt{in}\ u$

Figure 2.5: The syntax of $\mathcal{N}_{\texttt{let}}$.

Original Language: $\mathcal{N}_{\texttt{let}}$

Embedding Language: Coq

Embedding Domain:

```
Inductive mterm : Set :=
| Lit (n : nat)
| Plus  (t u : mterm)
| Minus (t u : mterm)
| Let (t : mterm)
      (u : nat -> mterm).
```

Semantic Domain:

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

Embedding Rules:

$$\begin{aligned}
[\![x]\!]_\rho &= \texttt{Lit}\ (\rho\ x)\\
[\![n]\!]_\rho &= \texttt{Lit n}\\
[\![t+u]\!]_\rho &= \texttt{Plus}\ [\![t]\!]_\rho\ [\![u]\!]_\rho\\
[\![t-u]\!]_\rho &= \texttt{Minus}\ [\![t]\!]_\rho\ [\![u]\!]_\rho\\
[\![\texttt{let}\ x = t\ \texttt{in}\ u]\!]_\rho &= \texttt{Let}\ [\![t]\!]_\rho\ \texttt{(fun x =>}\ [\![u]\!]_{\rho \bigcup \{x \mapsto \texttt{x}\}}\texttt{)}
\end{aligned}$$

(x is a fresh variable in Coq)

Interpretation:

```
Fixpoint interp (t : mterm) : nat :=
  match t with
  | Lit n => n
  | Plus t u => interp t + interp u
  | Minus t u => interp t - interp u
  | Let t u => interp (u (interp t))
  end.
```

Figure 2.6: A mixed embedding from $\mathcal{N}_{\texttt{let}}$ to our `mterm` type in Coq.

Honsell et al., 2001; Pfenning and Elliott, 1988; Prinz et al., 2022; Washburn and Weirich, 2008). In this part, I illustrate the style of Chlipala (2021). To make our language $\mathcal{N}$ more interesting for a mixed embedding, we add variables and let bindings to it. The let bindings are call-by-value. We name this new language $\mathcal{N}_{\texttt{let}}$. We show the syntax of $\mathcal{N}_{\texttt{let}}$ in Fig. 2.5.

We would like to follow the deep-embedding style here because it allows us to reason about syntactic structures of original programs. However, as discussed earlier, variable bindings pose a challenge for deep embeddings. To avoid the trouble of defining and reasoning about variable bindings, we can make let bindings shallow.

The mixed embedding of $\mathcal{N}_{\texttt{let}}$ is shown in Fig. 2.6. Our embedding domain is a custom inductive type called `mterm` (stands for a "mixed term") that adds a `Let` constructor compared with `term` (Fig. 2.4). We do not need a constructor for variables because we use Coq's variables instead—The key to implement this is in the second parameter of `Let`: we use Coq's native functions to represent the scope of let bound variables.[2] For example, $\texttt{let}\ x = 1 + 2\ \texttt{in}\ x + 3$ is represented by:

```
Let (Plus (Lit 1) (Lit 2))
    (fun x => Plus (Lit x) (Lit 3)).
```

To embed $\mathcal{N}_{\texttt{let}}$ using `mterm` in Coq, we augment our embedding rules $[\![\cdot]\!]$ by adding a set of mappings ρ that tracks which variable in $\mathcal{N}_{\texttt{let}}$ corresponds to which variable in Coq. Whenever there is a let binding, we create a fresh variable in Coq and add that to ρ, as shown in the embedding rules in Fig. 2.6.

Our mixed embedding reap certain benefits of both shallow and deep embeddings. For example, we can check if a mixed term representing a term in $\mathcal{N}_{\texttt{let}}$ contains any subtractions, similar to the deep embedding shown earlier:

```
Fixpoint no_minus (t : mterm) : bool :=
  match t with
  | Lit _ => true
  | Plus t u => no_minus t && no_minus u
  | Minus _ _ => false
  | Let t u => no_minus t && no_minus (u 0)
  end.
```

For `Let`'s second parameter `u : nat -> mterm`, we simply pass any natural number to it (in this case, I pass `0`), because $\mathcal{N}_{\texttt{let}}$ does not have any terms that change the control flow (*i.e.,* there is no if statement or pattern matching, *etc.*). In the meantime, our mixed embed-

[2]The technique is very similar to the idea behind higher-order abstract syntax (HOAS) (Harper et al., 1987; Pfenning and Elliott, 1988). A key difference is that HOAS uses functions on the deep embedding of terms, while we use the shallow embedding of terms here, as shown by the input type of the second parameter of `Let`.

ding avoids defining or reasoning about variable bindings or capture-avoiding substitutions, similar to a shallow embedding.

There are, however, a few problems with our mixed embeddings as well. First, our mixed embedding is not as modular as the deep embedding in Fig. 2.4. This is due to that the second parameter of `Let` is a function on `nat`—this constraints that our interpretation of an `mterm` to return a `nat`. As a consequence, our mixed embedding is restricted to a semantic domain of `nat` (Fig. 2.6).

The second problem is that when we make certain parts of an embedding shallow, we might make our embedding domain more expressive than the original language, which can be a problem if we would like to reason about properties that rely on the original language to *not* be very expressive. For example, for any term t in $\mathcal{N}_{\texttt{let}}$, `let` $x = 0$ `in` t and `let` $x = 1$ `in` t have the same number of $+$ or $-$ operations. But we cannot prove this property with our mixed embedding. Indeed, consider the following two `mterm`s:

```
Example example1 :=
  Let (Lit 0)
      (fun x => if x =? 0 then Lit 0
                else (Plus (Lit x) (Lit 2))).

Example example2 :=
  Let (Lit 1)
      (fun x => if x =? 0 then Lit 0
                else (Plus (Lit x) (Lit 2))).
```

These two `mterms` are called *exotic terms* because they do not correspond to any terms in $\mathcal{N}_{\texttt{let}}$. However, we cannot show the nonexistence of these exotic terms in Coq because `mterms` are the only representation of $\mathcal{N}_{\texttt{let}}$ we have available in Coq.

There are other alternatives to our mixed embedding here. One example is parametric higher-order abstract syntax (PHOAS) (Chlipala, 2008; Washburn and Weirich, 2008). Compared with our mixed embedding, PHOAS also allows us to define and reason about functions like `no_minus`, it does not limit its semantic domain, and it does not introduce exotic terms. However, PHOAS does not offer a free implementation of capture-avoiding substitutions, which is the main reason we present the mixed embedding of Chlipala (2021) rather than PHOAS here.

In summary, there are many styles of mixed embeddings that allows us to reap the benefits of both shallow and deep embeddings. However, *where to draw the line* between shallow and deep embeddings can greatly impact what properties can be easily proven or what properties can even be proven, and is a design choice that requires weighing various trade-offs.

2.4. Using Monads

As discussed earlier, functional programming languages usually "hide" evaluation strategies such as calling conventions under the hood. To reason about "how" using functional languages, we need a way to make evaluation strategies *explicit*. The typical way of "reifying" evaluation strategies is using *monads*. In this section, we demonstrate the way of *using monads* as embedding domains to reify evaluation strategies of original languages.

Monads are a concept in category theory (Barr and Wells, 1990) and they are introduced to programming languages research by Moggi as a way to represent "notions of computation" in a semantics (Moggi, 1991). Based on Moggi's work, Wadler (1992) proposes two ways of translating a program to a monadic program that corresponds to the call-by-value and call-by-need semantics, respectively.

Later, Petricek (2012) finds that these two ways can be unified under *one* abstract translation strategy. In this part, we reformulate Petricek's translation strategy for call-by-value and call-by-need semantics as embedding rules.

First, we recall the definition of a monad in Coq:

```
Class Monad (F : Type -> Type) `{Functor F}:=
  { ret : forall {A}, A -> F A ;
```

```
  bind : forall {A B}, F A -> (A -> F B) -> F B }.
```

A `Monad` is required to define two operations: a `ret` operation that "wraps" a value inside the monad, and a `bind` operation that "connects" two monadic computations together. Noticeably, the type of the `bind` operation "forces" its first explicit parameter of type `F A` to be evaluated to pass a value to its second explicit parameter of type `A -> F B`, so it can be viewed as a way of reifying evaluation orders. For convenience, we use notation `x <- e1 ; e2` to represent `bind e1 (fun x => e2)`, similar to Haskell's `do` notation.

To illustrate the embedding adapted from Petricek (2012), we reuse the call-by-value language $\mathcal{N}_{\texttt{let}}$ as one of our original languages; the other one is $\mathcal{N}^n_{\texttt{let}}$, a call-by-name variant of $\mathcal{N}_{\texttt{let}}$—the only difference between $\mathcal{N}^n_{\texttt{let}}$ and $\mathcal{N}_{\texttt{let}}$ is that the let bindings of $\mathcal{N}^n_{\texttt{let}}$ is call-by-name. We again use Coq as our embedding language. We use an abstract monad `M : Type -> Type` as our embedding domain: a closed term in $\mathcal{N}_{\texttt{let}}$ is translated to type `M nat`; a term with one open variable is translated to type `M nat -> M nat`; a term with two open variables is translated to type `M nat -> M nat -> M nat`, and so on. The unified embedding rules for both $\mathcal{N}_{\texttt{let}}$ and $\mathcal{N}^n_{\texttt{let}}$ are shown in Fig. 2.7. The key to implement a call-by-value or a call-by-name semantics lies in the definition of `malias`. The function has the same type under both semantics but with different implementations, as shown in Fig. 2.7. For a call-by-value semantics, it "forces" the computation of a parameter and returns the result; for a call-by-name semantics, it simply returns the computation.

Is our embedding a shallow, deep, or mixed embedding? It depends on the specific monads we use to instantiate `M`. Most of the common monads we have seen such as the list monads or writer monads are computational—the embedding would be a shallow embedding if we use any of them as the embedding domain. However, we can also instantiate the embedding as a deep or mixed embedding, *e.g.,* using free monads (Kiselyov and Ishii, 2015). We will revisit the use of free monads in Chapter 3 and Chapter 5.

Original Language:
$\mathcal{N}_{\texttt{let}}$, $\mathcal{N}^{n}_{\texttt{let}}$

Embedding Domain:

```
Variable M : Type -> Type.
Context `{Monad M}.
Parameter malias : forall {A},
  M A -> M (M A).
```

Embedding Language: Coq

Embedding Rules:

$$
\begin{aligned}
[\![x]\!]_\rho &= \texttt{ret}\ (\rho\ x)\\
[\![n]\!]_\rho &= \texttt{ret n}\\
[\![t+u]\!]_\rho &= \texttt{t <- } [\![t]\!]_\rho\texttt{; u <- } [\![u]\!]_\rho\texttt{; ret (t + u)}\\
&\quad \text{(}\texttt{t}\text{ and }\texttt{u}\text{ are fresh variables in Coq)}\\
[\![t-u]\!]_\rho &= \texttt{t <- } [\![t]\!]_\rho\texttt{; u <- } [\![u]\!]_\rho\texttt{; ret (t - u)}\\
&\quad \text{(}\texttt{t}\text{ and }\texttt{u}\text{ are fresh variables in Coq)}\\
[\![\texttt{let } x = t \texttt{ in } u]\!]_\rho &= \texttt{x <- malias } [\![t]\!]_\rho\texttt{;}\\
&\quad [\![u]\!]_{\rho \bigcup \{x \mapsto \texttt{x}\}}\ \texttt{x}\\
&\quad \text{(}\texttt{x}\text{ is a fresh variable in Coq)}
\end{aligned}
$$

Implementation of `malias` for $\mathcal{N}_{\texttt{let}}$ (call-by-value):

```
Definition malias {A} (m : M A) : M (M A) := bind m (fun x => ret (ret x)).
```

Implementation of `malias` for $\mathcal{N}^{n}_{\texttt{let}}$ (call-by-name):

```
Definition malias {A} : M A -> M (M A) := ret.
```

Figure 2.7: An embedding from $\mathcal{N}_{\texttt{let}}$ to monad `M` in Coq.

Petricek also proposes an implementation of `malias` that defines call-by-need semantics (Petricek, 2012). However, the implementation is in Haskell and it relies on Haskell's effectful `ST` monad transformer (Launchbury and Jones, 1995; Svenningsson, 2011), a feature that is not supported by pure languages such as Coq, so we do not show that implementation here. However, we will show a revision of Petricek's uniform embedding rules that supports reasoning about the computational cost of a call-by-need semantics in Section 4.7.

2.5. Embeddings *vs.* Denotational Semantics

Denotational semantics (Scott, 1970; Scott and Strachey, 1971) is a method of defining a program's semantics using mathematical objects. The mathematical objects we use in programming languages research are typically lattices or domains (Scott, 1976, 1982). In its broader sense, we can view the "essence" of a Coq embedding as a denotational semantics: a Coq program that can be viewed as mathematical object. In this dissertation, we make a distinction between embeddings and denotational semantics by their practical differences discussed below:

First, a denotational semantics is usually used to describe the *semantics* of a language. On the other hand, an embedding is used to describe a program in a way that is amenable for mechanized reasoning about certain properties. These properties might contain only syntactic properties, or both semantic and syntactic properties.

Second, the syntax of a language is typically defined in the *same host language* as the defined language's denotational semantics, but this is not the case in an embedding other than deep embeddings. Therefore, it is not a problem to state or reason about properties related to a program's syntactic structure in a denotational semantics, but it is important to consider the issue in an embedding. For the same reason, exotic terms are a problem to embeddings, but not to denotational semantics, because we can use the syntax to distinguish original and exotic terms.

Third, a denotational semantics is a *total* mapping from well-formed syntax to semantics, but an embedding does not need to be so. The embedding rules in an embedding can be partial, so that we only deal with a subset of the language features. When we encounter an expression that cannot be translated by any embedding rules, we can either require *human intervention* (Danielsson, 2008; Handley et al., 2020; Spector-Zabusky et al., 2018, 2019) or just fail to embed the program. Furthermore, it is also fine if the embedded domain does not model the semantics of the original language 100%—as long as the embedding is useful for proving properties that we care about.

CHAPTER 3

Experience with Embeddings

Embeddings play important roles in two projects that I have contributed to during my Ph.D. journey. Not surprisingly, my experience in these two projects has significantly influenced my views on embeddings and has led me to works in this dissertation. In this chapter, I summarize these two projects and discuss how they formulate my views on embeddings.

3.1. A Shallow Embedding of Haskell in Coq

The first project is about mechanized reasoning about Haskell code in Coq. The project is based on `hs-to-coq`, a tool initially developed by Spector-Zabusky, Breitner, Rizkallah, and Weirich (2018). One important characteristic of `hs-to-coq` is that it embeds Haskell in Coq using a *shallow embedding.* The list of embedding rules of `hs-to-coq` is too large for me to show it in this dissertation. Instead, I show an example of Haskell's source code and its shallow embedding in Coq using `hs-to-coq` in Fig. 3.1 and Fig. 3.2, respectively.

One of the appeals of a shallow embedding is that it allows mechanized reasoning to reuse existing theorems, tactics, *etc.* of the embedding language. Indeed, this is the main reason that Spector-Zabusky et al. selected a shallow embedding. They wanted to reason about large and complex Haskell programs while reusing Coq's rich ecosystem (Appel, 2022; Ringer et al., 2019a) to facilitate this task.

However, one of the main challenges of using a shallow embedding in `hs-to-coq` is that there are many discrepancies between Haskell and Coq. For example, Haskell is call-by-need and its data types are coinductive; Haskell accepts non-terminating functions or complex non-structural recursions; you can invoke effects in Haskell; the integers in Haskell are bounded, *etc.* The key that contributes to `hs-to-coq`'s success is that it takes a pragmatic approach that does *not* try to be total or absolutely faithful to Haskell's semantics: its embedding rules focus on *inductive* data types and *total, terminating* functions, "where the semantics of lazy and strict evaluation, and hence of Haskell and Coq, coincide" (Danielsson

```haskell
module Control.Applicative.Successors where

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

getCurrent :: Succs t -> t
getCurrent (Succs x _) = x

getSuccs :: Succs t -> [t]
getSuccs (Succs _ xs) = xs
```

Figure 3.1: Part of the source code of Haskell's `Successors` library.

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

Arguments succs {_} _ _.

Definition getCurrent {t : Type} : Succs t -> t :=
  fun '(succs x _) => x.

Definition getSuccs {t : Type} : Succs t -> list t :=
  fun '(succs _ xs) => xs.
```

Figure 3.2: The Coq code that is automatically translated from the code in Fig. 3.1 using `hs-to-coq`.

et al., 2006; Spector-Zabusky et al., 2018). When a language feature of Haskell that is not supported by `hs-to-coq`'s embedding rules shows up, a language called `edits` (Spector-Zabusky, 2021, Chapter 4 & Chapter 8) is used so that human can guide its translation.

I joined the project around 2017 and contributed to two studies where we applied `hs-to-coq` to real-world examples. The first example is the `containers` library, a Haskell library that encodes data structures for finite sets and maps (Breitner et al., 2018, 2021). The second example is parts of the Glasgow Haskell Compiler (GHC), an industrial-strength compiler for Haskell (Spector-Zabusky et al., 2019).

Verifying `containers` Our first study of applying `hs-to-coq` to a real-world example is verifying Haskell's `containers` library (Breitner, Spector-Zabusky, Li, Rizkallah, Wiegley, Cohen, and Weirich, 2021). The library is "the third-most depended-on package of the Haskell package repository Hackage," (Breitner et al., 2021, Section 2) and it is a highlyoptimized code base with more than a decade's history of performance tuning (Breitner et al., 2021, Section 2.3). We show that it is possible to apply a shallow embedding to embed a significant portion of this library in Coq, and we show the functional correctness of a representative set of its commonly used functions based on this shallow embedding (Breitner et al., 2021, Section 3). The specification itself is drawn from many different sources. It describes the library's complex behaviors in detail and is "connected to both implementations and clients" (Breitner et al., 2021, Section 4). We did not find any bugs in reasoning about `containers`, but our verification provided some new insights to both implementing and verifying data structures in `containers` (Breitner et al., 2021, Section 6). In this study, I focused on verifying a finite set library called `Data.Set`. I also helped with other parts of the study, such as fixing issues in the `hs-to-coq` tool and co-authoring papers on this verification effort (Breitner et al., 2018, 2021).

Our experience shows that shallow embeddings in an interactive theorem prover with a rich ecosystem are indeed a practical approach for mechanized reasoning about large-scale existing programs. Indeed, the `containers` library is no toy example: the relevant modules of the `containers` library "contain 325 functions and 41 type class instances, written in 4096 lines of code (excluding comments and blank lines)" (Breitner et al., 2021, Section 3). In total, we verified 2291 lines of Haskell and each line of Haskell roughly required 9.0 lines of proofs (Breitner et al., 2021, Section 3). Being able to use existing interface such as `FSetInterface`,[3] theorems such as those derived from `FSetInterface`,[4] and tactics such as `lia` (Besson, 2006) for reasoning about the weights in weight-balanced trees (Adams, 1992; Nievergelt and Reingold, 1973) greatly helped our verification effort. Among much great verification effort on Haskell (Abel et al., 2005a,b; Christiansen et al., 2019; Dybjer et al., 2004; Dylus et al., 2019; Hallgren, 2003; Vazou, 2016; Vazou et al., 2013, 2014, 2018b; Vytiniotis et al., 2013), our work is the only one that verifies a code base at this scale with a

[3] https://coq.inria.fr/library/Coq.FSets.FSetInterface.html
[4] https://coq.inria.fr/library/Coq.FSets.FSetFacts.html

rich specification[5]—another evidence that our pragmatic approach with shallow embeddings helped greatly.

Verifying GHC In our second study, we tried to reason about a code base of an even larger scale—but this time, we only focused on a small part of it (Spector-Zabusky, Breitner, Li, and Weirich, 2019). **The main research question was:** can we still benefit from mechanized reasoning if we only consider a small part of a large system? We chose to reason about some of GHC's optimizations based on its intermediate language called Core (Eisenberg, 2020; Sulzmann et al., 2007). GHC itself is very large and complex, and all its modules are intertwined. To manage the scale of our verification effort, we needed a way to embed a particular part of GHC while "abstracting" all others. We achieved this by utilizing the `edits` language mentioned earlier to create a *documented and mechanized* "formalization gap" between the original program and the embedded program (Spector-Zabusky et al., 2019, Section 5 & Section 6). Our study showed that it is indeed useful to apply mechanized reasoning even when there is a large formalization gap: our verification effort exhibited a bug in GHC's code and a bug in its comment; furthermore, our specification also offered a documented and rigorous description of the invariants GHC maintained (Spector-Zabusky et al., 2019, Section 4). My contribution in this study focused on verifying data structures that represents sets of variables in Core. Similar to the previous study, I also helped with other parts including fixing issues in the `hs-to-coq` tool and co-authoring a paper on this verification effort (Spector-Zabusky et al., 2019).

Our experience suggests that we should look at the formalization gap between original and embedding languages from a different perspective. The gap has always been considered a critical issue with shallow embeddings. However, our study showed that, by carefully managing the formalization gap in a documented and mechanized way, we were able to control the scale of verification effort to mechanically reason about complex real-world software

[5] Vazou et al. (2013) is perhaps the closest, as they verified the `Data.Map` module, which is the finite map module in `containers`. However, their specification only describes the orders of elements in the tree data structure underlying the module. We compare our work and Vazou et al. (2013) in more detail in Breitner et al. (2021, Section 8.1).

systems at scale. **The experience again demonstrates** the usefulness of shallow embeddings, as we were able to verify parts of an industrial-strength compiler based on shallow embeddings.

However, there are a few limitations with the shallow embedding used by `hs-to-coq` as well. Most notably, we cannot use `hs-to-coq` to reason about properties about "how" such as the computation cost or effects. These limitations are one of the major motivations behind my works in Chapter 4 and Chapter 5.

3.2. From C to ITrees: From a Deep Embedding to a Mixed Embedding

The other project that forms my view on embeddings is building a verified networked server. The project is part of grand expedition called the Science of *Deep Specification* (Appel et al., 2017), which aimed to connect disparate verification and testing tools to build fully verified software stack. In our project, we aimed to build a server that runs on CertiKOS, a verified operating system (Gu et al., 2019), and the server is compiled using CompCert, a verified C compiler (Leroy, 2009). There were simultaneously two research questions in this project: (1) what is the right way to specify a server's behavior over the network, and (2) how to connect disparate verification and testing works (more on these tools shortly).

In our first work (Koh, Li, Li, Xia, Beringer, Honoré, Mansky, Pierce, and Zdancewic, 2019), we built a simple "swap server" (Koh et al., 2019, Section 2). The server was compiled using CompCert and it was built on system calls provided by CertiKOS, which contained an unverified TCP (Transmission Control Protocol) implementation along with its axiomatized specification. In addition, we used VST, a framework based on CompCert and Hoare-style separation logic (Appel et al., 2014) to specify and reason about the server (Koh et al., 2019, Section 5); we used QuickChick, a property-based testing tool (Lampropoulos and Pierce, 2021; Paraskevopoulou et al., 2015) to test our implementation as well as our specification (Koh et al., 2019, Section 6). On the top level, we wrote a specification that described a server's observable behavior over a network in a straightforward way; while in the lower levels, we described the server's behavior in terms of low-level operations such as system calls; to connect the specifications at higher and lower levels, we developed a special relation called *network refinement* that considers network re-orderings, *etc.* (Koh et al., 2019, Section 4). To tie all these different tools and different levels of specification together, we proposed a novel data structure called *interaction trees* (Koh et al., 2019, Section 3). I have been involved in the project since the beginning of it. I contributed to many aspects of the work, including many early explorations using other styles of programming logics and data structures, integrating interaction trees and Hoare-style separation logic, working on the mechanized proofs, and co-authoring the paper that describes this project (Koh et al., 2019), *etc.* Later, Hengchu Zhang joined the project and scaled up the verified "swap server" to a verified key-value server (Zhang, Honoré, Koh, Li, Li, Xia, Beringer, Mansky, Pierce, and Zdancewic, 2021).

One important thing we learned from the experience is the usefulness of interaction trees. Indeed, our collaborators Xia, Zakowski, He, Hur, Malecha, Pierce, and Zdancewic later expanded on this idea and developed a library for interaction trees (Xia et al., 2020), which was subsequently used in many projects (Foster et al., 2021; Lesani et al., 2022; Li et al., 2021c; Silver and Zdancewic, 2021; Ye et al., 2022; Yoon et al., 2022; Zakowski et al., 2021).

But why would interaction trees make a good data structure for connecting different verification and testing tools, and different levels of specification? Intuitively, this is because interaction trees are a low-level but general abstraction of effects and computations. We can use them to represent many kinds of effects (Xia et al., 2020, Section 3), we can *run* them by interpreting those effects (Xia et al., 2020, Section 3), and we can use them to model iterations and general recursions (Xia et al., 2020, Section 4). We cover these in more detail in Koh et al. (2019, Section 3). Xia et al. provide a more in-depth dive into the theory of interaction trees from the perspective of a denotational semantics in Xia et al. (2020).

In this section, however, I'd like to revisit the question of "why interaction trees" from a new perspective—a perspective of embeddings. In our task of verifying a networked server

```
Inductive expr : Type :=
  | Econst_int: int -> type -> expr       (**r integer literal *)
  | Econst_float: float -> type -> expr   (**r double float literal *)
  | Econst_single: float32 -> type -> expr (**r single float literal *)
  | Econst_long: int64 -> type -> expr    (**r long integer literal *)
  | Evar: ident -> type -> expr           (**r variable *)
  | Etempvar: ident -> type -> expr       (**r temporary variable *)
  | Ederef: expr -> type -> expr (**r pointer dereference (unary [*]) *)
  | Eaddrof: expr -> type -> expr         (**r address-of operator ([&]) *)
  | Eunop: unary_operation -> expr -> type -> expr  (**r unary operation *)
  | Ebinop: binary_operation -> expr -> expr -> type -> expr (**r binary
                                                               operation *)
  | Ecast: expr -> type -> expr   (**r type cast ([(ty) e]) *)
  | Efield: expr -> ident -> type -> expr (**r access to a member of a struct
                                               or union *)
  | Esizeof: type -> type -> expr         (**r size of a type *)
  | Ealignof: type -> type -> expr.       (**r alignment of a type *)
```

Figure 3.3: CompCert's embedding domain for expressions in Clight.

written in C, our first step is to embed C in Coq. This step is achieved by using CompCert, which models a subset of C called Clight (Blazy and Leroy, 2009) using a deep embedding. I show a code snippet of the embedding domain CompCert 3.10 uses to embed expressions in Clight in Fig. 3.3.[6] Such a deep embedding is crucial to CompCert as CompCert uses it to prove the correctness of transformations in its compiler. However, such an embedding would not be interesting for other verification tools such as QuickChick, nor would it be a good way for describing a server's observable behavior over a network, because it is too specific to the syntax of Clight.

A shallow embedding would not be a good fit, either. This is because a shallow embedding would be too specific to the semantics used by a particular tool or a particular level of specification. For example, QuickChick requires its semantics to be *executable* but it does not require the semantics to be defined within Coq—there is no issue in using QuickChick to invoke a foreign function such as running an actual server written in C. On the other hand, our high-level specification demands our semantics used for reasoning to be *nondeterministic*,

[6]https://github.com/AbsInt/CompCert/blob/v3.10/cfrontend/Clight.v

```
CoInductive itree (E : Type -> Type) (R : Type) :=
| Ret (r : R)
| Vis {X : Type} (e : E X) (k : X -> itree E R)
| Tau (t : itree E R).

Definition ret {E R} : R -> itree E R := Ret.
Definition bind {E R S} (t : itree E R) (k : R -> itree E S) : itree E S :=
  (cofix bind_ u := match u with
                    | Ret r => k r
                    | Tau t => Tau (bind_ t)
                    | Vis e k => Vis e (fun x => bind_ (k x))
                    end) t.

Definition trigger {E R} (e : E R) : itree E R := Vis e Ret.
```

Figure 3.4: The key definitions of interaction trees or `itree`s.

so our verification considers nondeterminism in an operating system and re-orderings in a network, and the semantics should definitely be defined within Coq.

These requirements call for a "compromise" between a deep and a shallow embedding, so they lead us to a mixed embedding, which is exactly what interaction trees offer. In the end of Section 2.3, I stated that a key design choice to make in a mixed embedding is *where to draw the line* between shallow and deep embeddings. Interaction trees provide a good answer to this question in the space of effectful computation—we embed pure computations "shallowly" and effects "deeply".

I show the key definitions of interaction trees or `itree`s in Fig. 3.4.[7] An interaction tree has two parameters: an *uninterpreted* effect data type `E : Type -> Type` and a return type `R : Type`. It has three constructors: a `Ret` constructor that "wraps" a pure computation inside it; a `Vis` constructor that represents a "visible" effect followed by a continuation; and a `Tau` constructor that represents a silent step. The `itree` data type is coinductive so it can be used to represent potentially non-terminating programs such as networked servers.[8] An `itree` is a monad, as shown by the `ret` and `bind` definitions in Fig. 3.4. In fact, interaction

[7]The figure is adapted from Koh et al. (2019, Fig. 6) and Xia et al. (2020, Section 2.1 & Fig. 3).
[8]More information about coinductive data types in Coq can be found in Chlipala (2019, Chapter 5).

```
Variant IO : Type -> Type :=
| Input : IO string
| Output : string -> IO unit.

Definition input : itree IO string := trigger Input.
Definition output (s : string) : itree IO unit := trigger (Output s).
```

Figure 3.5: An example of an effect data type.

trees are a *coinductive* variant of free monads (Kiselyov and Ishii, 2015). In addition, an `itree` also has a `trigger` function, which "lifts" an effect data type to an `itree`.

I show an example of an effect data type in Fig. 3.5.[9] The `IO` effect data type includes two types of effects: an `Input` that takes no parameter and returns a `string`, and an `Output` that takes a `string` as its parameter and returns a `unit`, which is Coq's builtin type that has only one inhabitant `tt`. Based on `IO` and with the help of the `trigger` function provided by `itree`, we can define two "effectful" functions `input` and `output`, also shown in Fig. 3.5. These functions, however, are *uninterpreted*. A separate interpreter is needed if we would like to reason about their semantics. In other words, the effect data type is a *deep embedding* of an effect.

In summary, interaction trees are a data structure that combines a shallow embedding of pure computation and a deep embedding of effects. The level of abstraction provided by such a mixed embedding is neither too specific to a syntax nor too specific to a semantics, which makes it general enough to model many effectful computation models. However, a follow-up question a reader might want to ask is: are interaction trees the only data structure suitable for this style of mixed embeddings? The question turns out to be a direct inspiration for my work in Chapter 5.

[9] The figure is adapted from Koh et al. (2019, Section 3).

CHAPTER 4

Reasoning about the Garden of Forking Paths

This chapter references previously published paper *Reasoning about the Garden of Forking Paths* (Li, Xia, and Weirich, 2021a), with adjustments to the flow and terminology. I also add Section 4.7 that relates our work to the embeddings of Petricek (2012) (Section 2.4). All the Coq definition, theorems, and proofs presented in this chapter can be found in a publicly available artifact (Li et al., 2021b). I contributed to most of the work in collaboration with my co-authors. The part on the equivalence between our clairvoyant embedding with clairvoyant call-by-value semantics was developed and written by my co-author Li-yao Xia.

The title of the original paper is a reference to the short story *The Garden of Forking Paths* by Jorge Luis Borges. In the short story, the garden of forking paths is "an infinite series of times, in a growing, dizzying net of divergent, convergent, and parallel times" (Borges, 1941).

4.1. Introduction

Lazy evaluation (Henderson and Morris, 1976), or the *call-by-need* calling convention, is a distinguishing feature in some functional programming languages—Haskell being the most notable example. Rather than evaluating eagerly, a lazy evaluator stores computations in a thunk and only evaluates the thunk when the data is needed. This feature avoids unneeded computation and enables better modularity in functional programming (Hughes, 1989). However, with convenience in expressiveness comes challenge in reasoning—especially so for cost analysis—because it's far less obvious if, when, and how much a computation is evaluated. We believe mechanized reasoning can help bring clarity to a semantics so intricate and subtle.

However, embedding a lazy program is difficult. The semantics for call-by-need evaluation is more complex than that of call-by-name or call-by-value, which can be described merely through substitution. Traditional presentations (Josephs, 1989; Launchbury, 1993) of call-by-need semantics are fundamentally stateful, based on heaps that contain thunks which must be updated during evaluation.

Rather than directly dealing with such complexity, we take advantage of a new way of modeling call-by-need: *clairvoyant call-by-value* (Hackett and Hutton, 2019). The key observation of this new model is that although *whether* a term gets evaluated matters, it doesn't matter *when* in run-time cost analysis. Therefore, instead of storing the computations in a thunk, the clairvoyant call-by-value model makes use of *nondeterminism* to evaluate the data in one branch and skip evaluation in another. Eventually, one successful branch of evaluation will faithfully model the result and cost of the call-by-need evaluation.

Based on clairvoyant evaluation, we propose a novel framework for embedding and reasoning about computation cost of lazy programs in Coq. Our framework is based on an annotated model similar to that of Danielsson (2008) and of Handley et al. (2020), but our work does not require human intervention to explictly model sharing under lazy evaluation.

4.2. Motivating example

We start by providing an informal overview of our approach on a small example that exhibits laziness. We use Coq to illustrate all examples so that all examples in this dissertation would be in the same language. Coq is not necessarily a lazy language, but we can imagine these Coq programs are obtained by translating from lazy Haskell programs using a tool like `hs-to-coq` (Section 3.1). For reasons that we will explain in Section 4.4, these examples are also translated to A-Normal Form (Sabry and Felleisen, 1992), but that doesn't matter so much here.

For a motivating example, consider the following program:

```
Definition p {a} (n : nat) (xs ys : list a) : list a :=
  let zs := append xs ys in
  take n zs.
```

```
Fixpoint append {a} (xs ys : list a) : list a :=
  match xs with
  | nil => ys
  | cons x xs1 => let zs := append xs1 ys in x :: zs
  end.

(* returns the prefix of xs of length n or xs when n > length xs. *)
Fixpoint take {a} (n : nat) (xs : list a) : list a :=
  match n, xs with
  | O, _ => nil
  | S _, nil => nil
  | S n1, x :: xs1 => let zs := take n1 xs1 in x :: zs
  end.
```

Figure 4.1: The pure functional definitions of `append` and `take`.

The functions `append` and `take` in this example are equivalent to their Haskell counterparts, and their definitions are shown in Fig. 4.1. These examples use Coq's inductively defined lists, which are a subset of Haskell's list type. Although working with infinite data types is another useful application of lazy evaluation, many algorithms manipulate only finite data structures (Okasaki, 1999). Hence, we believe inductive lists are representative of how lists are frequently used in practice even in Haskell.

To estimate the time it takes to evaluate a program, its cost, we can start by counting the number of steps in some operational semantics, or some proportional quantity. Let us count function calls informally.

Lazy evaluation leads us immediately to an impasse, because it is not even clear what it means to "run" a lazy program. Lazy programs are demand-driven, so we have to specify some model of "demand". A common working model is that lazy programs will be forced during the evaluation of a whole program, but it is not so practical to reason about the behaviors of arbitrary programs. A more useful approach is to start from a more familiar place: call-by-value. Indeed, programs under the call-by-value evaluation strategy have a relatively straightforward cost model. Laziness adds a twist to it: we might not need all of the result, in which case we allow ourselves to skip some computations.

With that new ability, we face the problem of deciding *which* computations to skip. This decision inherently depends on how much of the overall result will be needed. For concreteness, let us require all of our example list `take n (append xs ys)` to be evaluated. We start by evaluating `append xs ys`, unfolding the program in call-by-value. There are two cases to consider: the length of `xs` may be less than `n`, in which case we will fully evaluate `append xs ys` in `length xs + 1` calls—where the final `nil` takes one call. Or `length xs` may be greater than or equal to `n`, then we can stop after `n` calls, leaving the result of the next call "undefined". Either way, we will produce some partially defined list `zs` after at most `n` calls. We then let `take n zs` run to the end in at most `n + 1` calls, thus producing all elements of `take n (append xs ys)`, as we demanded initially. In total, that took at most `2 * n + 1` calls. In particular, that cost is independent of the length of `xs` or `ys`. That exemplifies one of the core motivations of laziness: you only pay for what you need.

That idea of "call-by-value with a twist" is made formal by the concept of *clairvoyant evaluation* (Hackett and Hutton, 2019).

Clairvoyant evaluation The key to formalizing the reasoning above is to view lazy programs as *nondeterministic* programs. Clairvoyant evaluation works in a way similar to nondeterministic automata, which choose one of multiple successor states by *"guessing"* the path to success. In our earlier reasoning, we evaluated `append xs ys` using a call-by-value semantics until we decided to stop at a point. *When* to stop was not decided by the state of the program, but by a "guess" based on the clairvoyant knowledge that we would only need `n` elements in the end. This intuition allows us to define a general semantics that the meaning of a lazy program comprises all of its nondeterministic evaluations and the meaning can be refined later in light of new external information.

The equivalence of clairvoyant evaluation to the natural heap-based definition of laziness of Launchbury (1993) was proved by Hackett and Hutton (2019): the cost of any execution in clairvoyant call-by-value is an upper bound of the cost in call-by-need, and there is some clairvoyant execution whose cost is actually the same as in call-by-need.

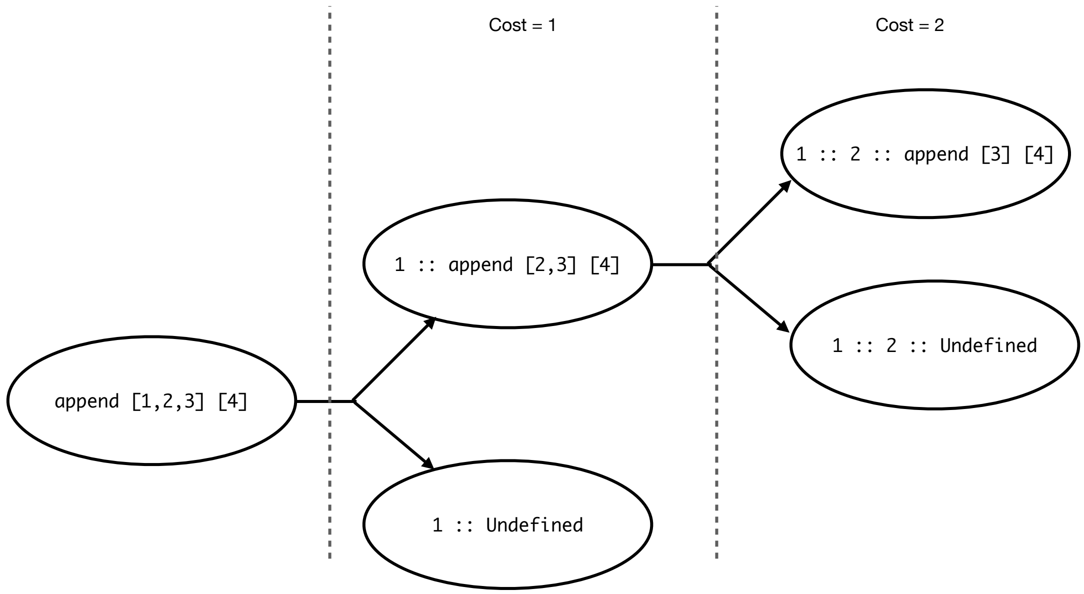

Figure 4.2: Parts of the nondeterministic clairvoyant call-by-value evaluation of the `append` function applied to two lists `[1, 2, 3]` and `[4]`.

Taking the `append` function as an example (recall its definition in Fig. 4.1), when it makes a recursive call, we fork the evaluation into two branches under clairvoyant evaluation: in branch (1), we perform the recursive call; and in branch (2), we skip that call. A skipped call yields a placeholder value as a result, which we call $\perp$ or `Undefined`.

Suppose that all future demands only require the first element of the result list, then branch (2) would suffice for offering that result. However, if a future demand requests more than that, branch (2) would fail to proceed because the requested data is `Undefined`. Therefore, branch (2) would get stuck and not yield any result at all. Fortunately, there would still be branch (1) to return the result. Furthermore, in branch (1), the `append` function may make another recursive call to itself, as long as the first argument list is not `nil`. In that case, this branch would be once again forked into two branches. This is illustrated by Fig. 4.2.

Although lazy programs are now interpreted nondeterministically, nondeterminism is used in a very controlled manner. The only choices we make are whether to perform a computation or to skip it. This means that the value of a program, if it exists, is still unique in some sense: the only possible changes are that parts of the value are replaced with `Undefined`.

If we know which branch leads to a successful evaluation for the `append` function, we can just look at that branch and add its cost to obtain the total cost of the program, which gives us a local reasoning methodology. Of course we cannot know this in advance, but there are some reasoning principles that can help.

A dual reasoning principle We would like to have a reasoning methodology that is both *local* and *modular*, just like what we would expect from functional programming. This means that we should be able to use some relations to specify the behaviors of each individual function. And when we want to reason about a program, we can just do that by composing the relations of its functions.

We use a dual reasoning principle to achieve the locality and modularity for clairvoyant call-by-value evaluation. First, we have a *pessimistic* specification that describes the behaviors of *all* of the function's nondeterministic branches. The pessimistic specification can offer us an accurate description of functional correctness under call-by-need evaluation. However, the specification is pessimistic because it does not rule out the branches that contain redundant steps and would not appear in an actual call-by-need evaluation.

To be more selective in those branches, we use an optimistic specification that describes the behaviors on a specific branch. The specification is optimistic because it can be used to specify a more accurate bound for the cost under call-by-need evaluation.

Figure 4.3 shows the relations among a clairvoyant evaluation, a pessimistic specification, and an optimistic specification. The tree in the middle of the figure represents the nondeterminism tree of clairvoyant evaluation. The gray nodes represent the end results of their branches. A pessimistic specification specifies the nodes in the red circle. And an optimistic specification specifies the node in the blue circle.

Getting back to `append`, its pessimistic specification states:

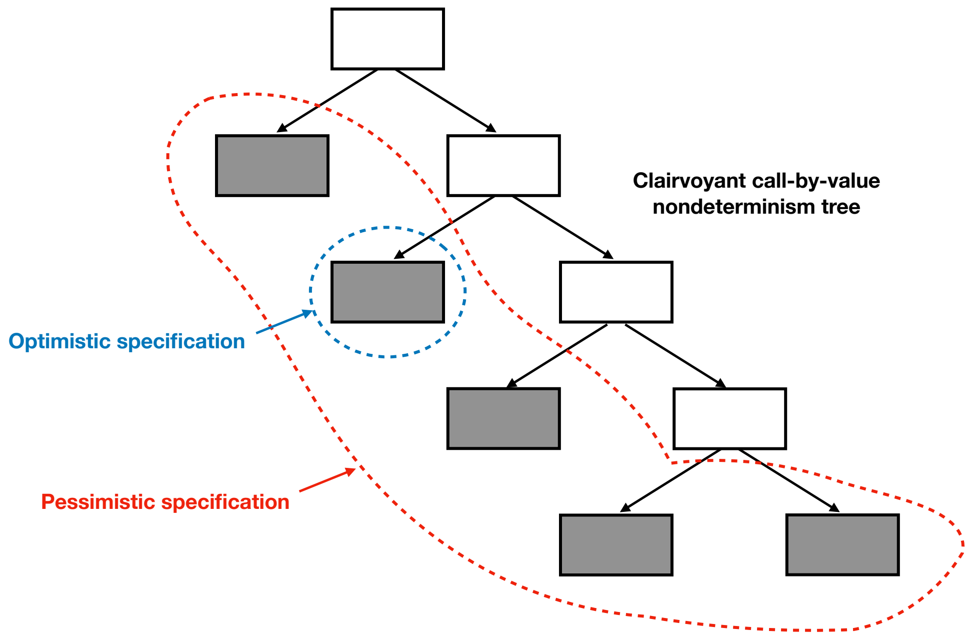

Figure 4.3: The relations among a clairvoyant evaluation, a pessimistic specification, and an optimistic specification.

> For *all* the nondeterministic branches of the `append` function, if the branch evaluates successfully, it will return a cost $c \in [1, \text{length } xs + 1]$.

For simplicity, we omit the functional correctness part of the specification here. The pessimistic specification only specifies a coarse range for `append`'s costs. If we want to reason about our earlier example `p`, we could not deduce that its cost would never go over `n` by reasoning about `append` and `take` abstractly using their specifications. Instead, we can only deduce that the upper bound of the cost is the length of `xs` and the length may be much larger than `n`.

For this sort of analysis, we need the optimistic specification of `append`:

> For any number $n \in [1, \text{length } xs]$ (or $n \in [1, \text{length } xs + 1]$ if xs does not contain any undefined part),[10] there *exists* a nondeterministic branch of the `append` function that evaluates successfully and returns a cost $c = n$.

[10]When xs does not contain any undefined part, `append` might go over the entire list and take one extra cost pattern matching on `nil`. This is the only case the cost of `append` will be bigger than the length of xs.

A major difference between the pessimistic and the optimistic specifications is that the latter does not only show a range of costs; it shows what exactly are the possible costs within this range. With the help of this specification, we can "pick" one branch where the possible cost, which might be much smaller than the length of xs, barely suffices for producing a list that `take` needs.

Both the pessimistic and optimistic specifications can be proved on the `append` function. And when reasoning about a larger program like `p`, all we need is to compose the specifications of `append` with the specifications of other functions like `take`.

The missing pieces So far, we have informally discussed our methodology. The main missing piece to develop in the rest of the chapter is to implement this methodology in the formal environment of the Coq proof assistant.

The first step is to associate the pure functional programs with versions that track execution costs and allow nondeterminism. It turns out that we can do that with monads using the techniques discussed in Section 2.4. In the next section, we define the *clairvoyance monad*, which distills the main features of clairvoyant evaluation. One attraction of the clairvoyance monad is its simplicity: its core definitions consist of merely 21 nonblank, noncomment lines of code in Coq. Then, we show embedding rules in the clairvoyance monad in Section 4.4. The final step, in Section 4.5, is to build a program logic for the clairvoyance monad that enables local and modular formal cost analysis in the style of optimistic and pessimistic specifications.

4.3. The Clairvoyance Monad

The clairvoyance monad (Fig. 4.4) is a lightweight abstraction that can express the semantics of instrumented lazy evaluation, suitable for cost analysis. Based on the ideas of clairvoyant evaluation (Hackett and Hutton, 2019), its simplicity is largely due to the absence of higher-order state commonly associated with laziness.

```
(* A computation that produces a value of type "a" after some number of ticks. *)
Definition M (a : Type) : Type := a -> nat -> Prop.

(* A computation that takes no time and yields a single value. *)
Definition ret {a} (v : a) : M a :=
  fun y n => (y, n) = (v, 0).

(* Sequence two computations and add their time. *)
Definition bind {a b} (u : M a) (k : a -> M b) : M b :=
  fun y n => exists x nx ny, u x nx /\ k x y ny /\ n = nx + ny.

(* A computation with unit cost. *)
Definition tick : M unit :=
  fun _ n => n = 1.

(* A thunk: either a known value or unused. *)
Inductive T (a : Type) : Type :=
| Thunk (x : a)
| Undefined.

(* Store a computation without evaluating it (zero cost). *)
Definition thunk {a} (u : M a) : M (T a) :=
  fun t n => match t with
             | Thunk v => u v n
             | Undefined => n = 0
             end.

(* Either continue computation with the value of a thunk or fail. *)
Definition forcing {a b} (t : T a) (f : a -> M b) : M b :=
  match t with
  | Thunk v => f v
  | Undefined => fun _ _ => False
  end.

(* Force a thunk. *)
Definition force {a} (t : T a) : M a := forcing t ret.
```

Figure 4.4: Core definitions of the clairvoyance monad `M`.

To model clairvoyant evaluation, we need a monad that can encode all the following three ingredients: (1) costs, (2) nondeterminism, and (3) failures on some nondeterministic branches. The clairvoyance monad is the simplest monad that meets the criterion.

A computation in the clairvoyance monad, of type `M a`, nondeterministically yields a value `v` of type `a` after some time `n`. A computation is defined as a set of such pairs `(v, n)`, encoded in Coq as a predicate `a -> nat -> Prop`. The `ret` of the monad yields the given value `v` with a computation cost of 0; it is a set containing only `(v, 0)`. The `bind` of the monad sequences computation by getting a result `(x, nx)` from the first operand `u`, and then feeding the value `x` to the continuation `k`, which then yields another result `(y, ny)`. The overall result is that latter value paired with the total cost `(y, nx + ny)`.

To track time, `tick` (Moran and Sands, 1999) is a computation with unit cost. This design follows the same rationale as Danielsson (2008): explicit ticks make the library lightweight and flexible to experiment with different cost models. For a given cost model, one can ensure that ticks are added consistently by an automatic embedding. We present such a set of embedding rules in Section 4.4.

The type of thunks `T` is structurally an `option` type. A thunk is either a known value, under the `Thunk` constructor, or it is `Undefined`. `Undefined` thunks are placeholders introduced when a computation is "skipped," because its result won't be needed.

For "laziness," we add two operations to create and force thunks. Intuitively, the `thunk` function stores a computation of type `M a` without evaluating it, and yields a *thunk*: a reference to that stored computation, of type `T a`. The `forcing` function looks up that reference to evaluate the corresponding computation and passes it to the continuation. This result is also stored in place of the computation, so that subsequent uses of `force` will not recompute the result.

In the clairvoyance model, `thunk`'s implementation nondeterministically chooses between (1) running the computation, yielding any one of its results in a `Thunk`, and (2) skipping it, yielding an `Undefined` result at no cost. The set of possible outcomes is implemented as a predicate: it accepts any pair `(Thunk v, n)` such that `(v, n)` is accepted by the given computation `u`, plus the pair `(Undefined, 0)`.

The `forcing` operation accesses the result stored in a thunk and passes it to a continuation. If there is indeed a value `Thunk v`, then `v` is the result, and we just pass it to the continuation `k`. We do not need to add any costs in this step: we already paid the cost of computing

$$\frac{\Gamma \vdash \texttt{t : M a} \qquad \Gamma, \texttt{(x : a)} \vdash \texttt{s : M b}}{\Gamma \vdash \texttt{let! x := t in s : M b}}\,\texttt{let!} \qquad \frac{\Gamma \vdash \texttt{t : M a} \qquad \Gamma, \texttt{(xA : T a)} \vdash \texttt{s : M b}}{\Gamma \vdash \texttt{let\textasciitilde{} xA := t in s : M b}}\,\texttt{let\textasciitilde}$$

$$\frac{\Gamma \vdash \texttt{f : a -> M b} \qquad \Gamma \vdash \texttt{xA : T a}}{\Gamma \vdash \texttt{f \$! xA : M b}}\,\texttt{(\$!)}$$

Figure 4.5: Typing rules for `let!`, `let~`, and `$!`

`v` on the thunk's creation. If the thunk is `Undefined`, then the computation fails: it has no result, as denoted by the empty set. Note that the only way to fail among the above five combinators is to use `forcing` and that `thunk` is the only way to produce thunks to apply `forcing` to. In spite of this underlying potential for failure, computations definable with these combinators always have at least one successful execution by never skipping a `thunk`. In that sense, our combinators adequately model a total language.

The empty computation `fun _ _ => False : M a` could also be added to the core definitions to represent partiality. In this chapter, we will stick to total functional programming (Turner, 2004).

When programming, we also rely on Coq notations for a few well-known monadic operations in addition to these core definitions. The infix notation ">>" abbreviates `bind` with a constant continuation:

```
Notation "t >> s" := (bind t (fun _ => s)).
```

In the clairvoyance monad, a common idiom is `tick >> t` to increase the cost of `t` by one.

Functions whose arguments are thunks are called *lazy*, in the sense that their arguments may not always be defined. Otherwise they are *eager*. Let us define the following notations, wrapping the monadic `bind` in more familiar syntax, akin to do-notation in Haskell and overloaded `let` in OCaml. The infix `$!` is named after a standard Haskell operator which makes a function strict. For reference, typing rules for these constructs are given in Fig. 4.5.

```
Notation "let! x  := t in s" := (bind t (fun x => s)).
```

```
Notation "let~ xA := t in s" := (bind (thunk t) (fun xA => s)).
Notation "f $! xA" := (forcing xA f).
```

We thus view the combinator `bind` as an "eager" `let!` construct, where the bound variable `x` is the result of the computation `t`. In contrast, a composition of `bind` and `thunk` provides a "lazy" `let~`, where `xA` is a thunk for the "delayed" computation `t`. In this chapter, variables of "lifted" types `T a` will be marked with a suffix "`A`," to constrast with variables of "unlifted" types `a`.

The definition of `T` has two important features. First, a thunk is merely a value, not a location in a "heap" as it would be in natural semantics of Launchbury (1993); this is key to the simplicity of our definitions. Second, the type constructor `T` can be used in inductive type declarations and plays well with the strict positivity condition imposed on them.[11] This will be essential in embedding recursive types in Section 4.4.

In the clairvoyance model, it is useful to think of thunks as a way to construct *approximations* (Scott, 1976). The type `T a` "lifts" a type `a` with an `Undefined` value which approximates all values of type `a`. Recursive types may contain nested thunks, thus defining rich domains of approximations. In the monad `M`, we view a lazy computation as producing an approximation of some pure, complete result; more precise approximations are more costly to compute. That structure will be made explicit in Section 4.5.

Remark The monad `M` coincides with the writer monad transformer (Liang et al., 1995) applied to the powerset monad `_ -> Prop`. This observation crisply summarizes the orthogonal roles of nondeterminism and accounting for time in the clairvoyance monad.

4.4. Shallow Embeddings in the Clairvoyance Monad

The clairvoyance monad provides us with an explicit model of laziness. To reason about the cost of programs in a lazy language—where laziness is implicit—we embed them in the clairvoyance monad. Our original language is a total, lazy calculus with folds, enabling

[11] https://coq.inria.fr/refman/language/core/inductive.html#strict-positivity

$$
\begin{array}{rrcl}
\textit{types} & \tau & ::= & \tau \to \tau \mid \textsc{list}\ \tau \mid \textsc{unit} \\
 & x, y, z \in & & \textit{variables} \\
\textit{terms} & t, u & ::= & x \mid \lambda x.\, t \mid t\ x \mid \textsc{let}\ x\ =\ t\ \textsc{in}\ u \\
 & & & \mid \textsc{Nil} \mid \textsc{Cons}\ x\ y \mid \textsc{foldr}\ t\ (\lambda x\, y.t)\ z
\end{array}
$$

$$
\frac{\Gamma \vdash x : \textsc{list}\ \tau_1 \qquad \Gamma \vdash t_l : \tau_2 \qquad \Gamma,\, y_1 : \tau_1,\, y_2 : \tau_2 \vdash t_n : \tau_2}{\Gamma \vdash \textsc{foldr}\ t_l\ (\lambda y_1\ y_2.\ t_n)\ x : \tau_2}\ \textsc{foldr}
$$

Figure 4.6: Syntax and typing rules for $\lambda_{\textsc{foldr}}$.

$$
\begin{aligned}
[\![\tau_1 \to \tau_2]\!] &= \mathtt{T}\ [\![\tau_1]\!] \to \mathtt{M}\ [\![\tau_2]\!] \\
[\![\textsc{list}\ \tau]\!] &= \mathtt{listA}\ [\![\tau]\!]
\end{aligned}
$$

```
Inductive listA (a : Type) : Type :=
  NilA | ConsA (x1 : T a) (x2 : T (listA a)).
```

Figure 4.7: Embedding rules for types in $\lambda_{\textsc{foldr}}$ and the definition of `listA`.

structural recursion. Totality is arguably not a strong limitation for implementing many algorithms: in the context of complexity analysis, termination is a necessary condition for defining the cost of an algorithm. We name this language $\lambda_{\textsc{foldr}}$. Our embedding language is Coq and our embedding domain is the clairvoyance monad.

The syntax of $\lambda_{\textsc{foldr}}$ is summarized in Fig. 4.6. A primitive type of lists serves to illustrate how to translate algebraic data types, with structural recursion modeled by a FOLDR operator. This calculus is in A-normal form (Sabry and Felleisen, 1992) to streamline the embedding process by confining the bookkeeping of thunks to LET and FOLDR.

In embedding types of $\lambda_{\textsc{foldr}}$ (Fig. 4.7), function types $\tau_1 \to \tau_2$ are translated to function types $\mathtt{T}\ [\![\tau_1]\!] \to \mathtt{M}\ [\![\tau_2]\!]$. The argument is wrapped in a thunk, so functions may be defined on undefined inputs. And the result is, of course, a computation. The type of lists is translated to an inductive type where fields are wrapped in a thunk `T`. This type thus represents partially defined lists, which can be seen as *approximations* of actual lists.

$$
\begin{aligned}
[\![\text{LET } x \ = \ t \text{ IN } u]\!] &= \texttt{tick >> let\textasciitilde}\ x \ \texttt{:=}\ [\![t]\!]\ \texttt{in}\ [\![u]\!] \\
[\![x]\!] &= \texttt{tick >> force}\ x \\
[\![\lambda x.\ t]\!] &= \texttt{ret (fun}\ x\ \texttt{=>}\ [\![t]\!]\texttt{)} \\
[\![t\ x]\!] &= \texttt{tick >> let! f :=}\ [\![t]\!]\ \texttt{in f}\ x \\
[\![\text{NIL}]\!] &= \texttt{ret NilA} \\
[\![\text{CONS}\ x\ y]\!] &= \texttt{ret (ConsA}\ x\ y\texttt{)} \\
[\![\text{FOLDR}\ t_l\ (\lambda y_1\ y_2.\ t_n)\ x]\!] &= \texttt{foldrA}\ [\![t_l]\!]\ \texttt{(fun}\ y_1\ y_2\ \texttt{=>}\ [\![t_n]\!]\texttt{)}\ x
\end{aligned}
$$

Figure 4.8: Embedding rules for terms in λ_{FOLDR}.

```
Fixpoint foldrA' {a b} (n : M b) (c : T a -> T b -> M b) (x' : listA a) : M b :=
  tick >>
  match x' with
  | NilA => n
  | ConsA x1 x2 =>
    let~ y2 := foldrA' n c $! x2 in
    c x1 y2
  end.

Definition foldrA {a b} (n : M b) (c : T a -> T b -> M b) (x : T (listA a)) : M b :=
  foldrA' n c $! x.
```

Figure 4.9: Definition of the `foldrA` function used in embedding FOLDR.

In embedding terms of λ_{FOLDR} (Fig. 4.8), a well-typed term $t : \tau$ is translated to a Coq term $[\![t]\!]$: `M` $[\![\tau]\!]$. We pun source variables $x : a$ as target variables x : `T` $[\![a]\!]$. A `tick` is added uniformly in the interpretation of every non-value construct—this follows Hackett and Hutton (2019): it is assumed that those constructs will be implemented in constant time. In examples, we will simplify ticks further, as discussed later in this section.

Types guide the design of the term translation. The source `Let` corresponds to our lazy `let~`, creating thunks, while variable expressions $[\![x]\!]$: `M` $[\![\tau]\!]$ force the thunk denoted by the variable x : `T` $[\![\tau]\!]$.

The translation of FOLDR is defined in Fig. 4.9. A `tick` happens at every recursive call. Recursive calls are thunked using the `let~` construct, so that they may remain unevaluated if the `c` computation doesn't need them.

Recursion introduces a wrinkle in our translation. Generally, function arguments $x : a$ are lifted to $x : \texttt{T}\ [\![a]\!]$. However, recursive definitions in Coq must take an argument whose outer type constructor is defined using recursion, `listA a`, unlike the type of thunks `T (listA a)`. Thus the translated `foldrA` is merely a wrapper around the recursive function `foldrA'` where most of the work happens. Moreover, in `foldrA'`, the subterm `x2` is forced in continuation-passing style, using `forcing` (under the notation `$!`), in order to ensure that the recursive call to `foldrA'` is syntactically applied to a subterm of the initial list `x'`.

Equivalence with clairvoyant call-by-value Our embedding rules $[\![\cdot]\!]$ for terms in λ_{FOLDR} are also a cost-aware *denotational semantics* for λ_{FOLDR}. Formally, this semantics $[\![t]\!]$ is parameterized by an *environment* ρ that maps the free variables of t to semantic values. The denotation $[\![t]\!](\rho) : \texttt{M}\ [\![\tau]\!]$ is thus a set of cost-value pairs (n, v), with $n : \mathbb{N}$ and $v : [\![\tau]\!]$.

Xia proved the equivalence between our denotational semantics and *clairvoyant call-by-value* evaluation, the operational semantics of laziness introduced by Hackett and Hutton, which itself was proved equivalent to the natural semantics of Launchbury (1993). Clairvoyant call-by-value evaluation is defined as an inductive relation $t, h \Downarrow_n^{\text{CV}} u, h'$ expressing that the term t with an initial *heap* h evaluates, with cost n, to a *value term* u and a final heap h'. To state this equivalence, we extend our denotation function $[\![\cdot]\!]$ to these auxiliary syntactic constructs: a heap h, which maps variables to terms, denotes an environment $[\![h]\!]$, and a (syntactic) value term $u : \tau$, in an environment ρ, denotes a (semantic) value $[\![u]\!](\rho) : [\![\tau]\!]$.

We can now state an equivalence between our denotational semantics $[\![\cdot]\!]$ and the operational semantics $\Downarrow^{\text{CV}}$:

Theorem 1. *For any well-typed term t in λ_{FOLDR} and heap h, and for any value-cost pair (v, n), the following propositions are equivalent.*

1. $(v, n) \in [\![t]\!]([\![h]\!])$.

2. *There exists u and h' such that $t, h \Downarrow_n^{\text{CV}} u, h'$ and $v = [\![u]\!]([\![h']\!])$.*

The forward direction, (1) implies (2), states the *adequacy* theorem: the denotational semantics $[\![\cdot]\!]$ is a subset of the operational semantics $\Downarrow^{\mathrm{CV}}$. Conversely, the backward direction, (2) implies (1), states our *soundness* theorem: all evaluations by the operational semantics produce denoted cost-values. Together, these results prove that our semantics is equivalent to the operational semantics of Hackett and Hutton.

We have formalized both the denotational semantics $[\![\cdot]\!]$ and the operational semantics $\Downarrow^{\mathrm{CV}}$ and proved the equivalence theorem in Coq (Li et al., 2021b). The proof, which proceeds by induction on n for adequacy and on derivations of $\Downarrow^{\mathrm{CV}}$ for soundness, is straightforward thanks to the simplicity of the language, notably excluding general recursion. Most of the work is devoted to relating mutable heaps h in the operational semantics $\Downarrow^{\mathrm{CV}}$ to environments ρ in our denotational semantics $[\![\cdot]\!]$.

In addition to an operational semantics $\Downarrow^{\mathrm{CV}}$, Hackett and Hutton (2019) also presented a denotational cost semantics, which we can compare to ours. First, the cost semantics of Hackett and Hutton is defined for an untyped recursive calculus and our embedding is defined for a typed calculus with folds—guaranteeing termination. However, since the clairvoyance monad is based on the powerset monad `_ -> Prop`, we can also define a fixpoint operator (an example is Fig. 4.10):

```
Fix : ((a -> M b) -> (a -> M b)) -> (a -> M b)
```

Such an operator could be used for the denotational semantics of a general recursive lazy language, but at the cost of a more complex equivalence theorem. The issue is that the unfolding lemma `Fix F <-> F (Fix F)` assumes the monotonicity of `F`, adding a significant burden to using that operator. Without using `Fix`, we have only modelled a total language, as the source language we considered above is intended to be a subset of Coq. Many algorithms, in the functional programming literature especially, are defined using various forms of structural recursion, so they can be embedded in our framework.

```
Definition impl3 {a b c} (P P' : a -> b -> c -> Prop) : Prop :=
  forall x y z, P x y z -> P' x y z.

Inductive Fix {a b} (gf : (a -> M b) -> (a -> M b)) x y n : Prop :=
| MkFix (self : a -> M b) : impl3 self (Fix gf) -> gf self x y n -> Fix gf x y n.
```

Figure 4.10: Possible fixpoint combinator in the clairvoyance monad `M`.

Second, the denotations of Hackett and Hutton are cost-value pairs that inhabit a lattice to handle general recursion; they handle nondeterminism by joining all executions together. However, in the denotation of LET, the cost of evaluating the binding is discarded if the body of the LET does not depend strictly on the binding. In comparison, our semantics models computation using sets of pairs, so the cost of every nondeterministic path is preserved.

Key to the simplicity of our approach, the core operations of our clairvoyance monad (Section 4.3) are operations on sets; these operations do not rely on an abstract lattice structure for values. In exchange, our semantics is less well-behaved: `let~` expressions with unused thunks generate spurious approximations. We present a dual logic that disregards such uninformative approximations in Section 4.5.

An Example We show our translation in action on the example of `append` and `take`, illustrating a few pragmatic tweaks to our formalization above. The original program with pure functions in Fig. 4.1 is translated into the monadic program in Fig. 4.11. For simplicity, we retain the use of fixpoints instead of representing all recursion with `foldrA`.

These definitions use the `listA` type from Fig. 4.7. This type is the corresponding *approximation type* for Coq's `list`, and wraps every field in the thunk type constructor `T`.

The translation of recursive functions follows a similar structure to the definition of `foldrA` in the previous section, since `append` and `take` are in fact specialized list folds (`foldr`): their translations are wrappers for the recursive `append_` and `take_` where pattern-matching happens, and the recursive calls are guarded by thunks.

```
Fixpoint append_ {a : Type} (xs' : listA a) (ys : T (listA a)) : M (listA a) :=
  tick >>
  match xs' with
  | NilA => force ys
  | ConsA x xs1 =>
    let~ t := (fun xs1' => append_ xs1' ys) $! xs1 in
    ret (ConsA x t)
  end.

Definition appendA {a : Type} (xs ys : T (listA a)) : M (listA a) :=
  (fun xs' => append_ xs' ys) $! xs.

Fixpoint take_ {a : Type} (n : nat) (xs' : listA a) : M (listA a) :=
  tick >>
  match n, xs' with
  | O, _ => ret NilA
  | S _, NilA => ret NilA
  | S n1, ConsA x xs1 =>
    let~ t := take_ n1 $! xs1 in
    ret (ConsA x t)
  end.

Definition takeA {a : Type} (n : nat) (xs : T (listA a)) : M (listA a) :=
  take_ n $! xs.

Definition pA {a} (n : nat) (xs ys : T (listA a)) : M (listA a) :=
  tick >>
  let~ t := appendA xs ys in
  takeA n t.
```

Figure 4.11: The translated code of `append` and `take` from the pure version of Fig. 4.1.

We keep the primitive representation of certain types, such as `nat` in the definition of `take`, instead of using its Peano representation. The main reason to do so is that it makes the resulting program simpler by denoting "primitive" operations more directly. Although this is generally unsound for a language with pervasive laziness, this issue could be palliated by using a strictness analysis to ensure that variables of that type are never instantiated with $\perp$. Alternatively, we could consider an original language where both lifted and unlifted types coexist—Haskell is actually such a language, although unlifted types are not commonly used because GHC's strictness analysis is often good enough to enable optimizations.

The `append` function, of type `list a -> list a -> list a`, is translated to its approximate version `appendA`, of type `T (listA a) -> T (listA a) -> M (listA b)`. In other words, the arguments are put under thunks `T`, and the result is produced by an explicit computation `M`. This differs slightly with our formal translation where we simply translated a term $t : \tau$ to $[\![t]\!] : \mathtt{M}\ [\![\tau]\!]$. We can do away with that outer `M` because, typically, top-level functions are values.

Finally, we translate the bodies of the functions. To match the syntax of Fig. 4.6, we sequentialize expressions to ANF (Sabry and Felleisen, 1992) (if they are not already in this form), so that every computation happens at a LET binding. We then translate the ANF program to a monadic program, following Fig. 4.8.

Simplifying Ticks In our examples, we have simplified the translated code further to only keep a single `tick` at the head of source function bodies. This incurs a change to the cost of computations bounded by a multiplicative constant of the original cost. Considering that those costs are purely abstract quantities to begin with, this seems an acceptable trade-off to make the translated code more readable.

This simplification can be broken down in two steps. First, apply the following rewrite rules to "float up" every `tick` in every subexpression:

```
bind t (fun x => tick >> k x) = (tick >> bind t k)
```

```
thunk (tick >> u) <= (tick >> thunk u)
```

where the inequality `<=` means in this context that every execution `(x, nR)` on the right-hand side corresponds to an execution `(x, nL)` on the left-hand side with the same result but a lower or equal cost `nL <= nR`. The equality should be interpreted as extensional equality. That rewriting may increase the cost of programs, which is fine since we are eventually most interested in finding upper bounds on that cost. Second, once all ticks are as high in the program as they can be, we replace all consecutive ticks with a single one. The resulting "speed-up" of the computations is bounded by a constant multiplicative factor equal to the longest chain of ticks substituted that way.

4.5. Mechanized Reasoning

A guiding principle in designing our methodology is to have reasoning rules that are both *local* and *modular*. By local, we mean that we can reason about each function independently; and by modular, we mean we can reason about the whole program by composing the results of reasoning about its parts.

However, in doing so we face a challenge: clairvoyant call-by-value evaluation is an *over-approximation* of call-by-need evaluation: it contains nondeterministic branches that would not appear in an actual call-by-need evaluation. Therefore, to reason precisely about call-by-need execution, we not only need reasoning rules that are *general* enough to contain many nondeterministic results, but also *selective* enough to prune nondeterministic branches that contain redundant steps.

We address this challenge with a dual specification methodology. For *generality*, a *pessimistic* specification talks about the behaviors on all nondeterministic branches. For *selectiveness*, an *optimistic* specification describes the behavior of specific branches.

The optimistic and pessimistic specifications The definitions of the pessimistic specification and the optimistic specification are shown in Fig. 4.12. Both are parameterized by a *specification relation* `r : a -> nat -> Prop` which specifies a set of values and costs. A pessimistic specification states that all nondeterministic branches of the program `u` satisfy the relation `r`. On the other hand, the optimistic specification requires the existence of *at least one* nondeterministic branch satisfying the relation `r`.

We use the following notations to denote these two kinds of specifications:[12]

```
Notation " u {{ r }} " := (pessimistic u r).
Notation " u [[ r ]] " := (optimistic  u r).
```

[12] We omit the Coq notation levels in the code.

```
Definition pessimistic {a} (u : M a) (r : a -> nat -> Prop) : Prop :=
  forall x n, u x n -> r x n.

Definition optimistic {a} (u : M a) (r : a -> nat -> Prop) : Prop :=
  exists x n, u x n /\ r x n.
```

Figure 4.12: The definitions of the pessimistic and optimistic specifications.

$$\frac{r\ x\ 0}{(\texttt{ret}\ x)\ \{\{\ r\ \}\}}\texttt{ret} \qquad \frac{u\ \{\{\ \lambda x\ n.\ (k\ x)\ \{\{\ \lambda y\ m.\ r\ y\ (n+m)\ \}\}\ \}\}}{(\texttt{bind}\ u\ k)\ \{\{\ r\ \}\}}\texttt{bind}$$

$$\frac{r\ \texttt{tt}\ 1}{\texttt{tick}\ \{\{\ r\ \}\}}\texttt{tick} \qquad \frac{u\ \{\{\ r\ \}\} \qquad \forall x\ n, r\ x\ n \to r'\ x\ n}{u\ \{\{\ r'\ \}\}}\text{monotonicity}$$

$$\frac{r\ \texttt{Undefined}\ 0 \qquad u\ \{\{\ \lambda x.\ r\ (\texttt{Thunk}\ x)\ \}\}}{(\texttt{thunk}\ u)\ \{\{\ r\ \}\}}\texttt{thunk}$$

$$\frac{\forall x, t = \texttt{Thunk}\ x \to (k\ x)\ \{\{\ r\ \}\}}{(k\ \texttt{\$!}\ t)\ \{\{\ r\ \}\}}\texttt{forcing} \qquad \frac{u\ \{\{\ r\ \}\} \qquad u\ \{\{\ r'\ \}\}}{u\ \{\{\ \lambda x\ n.\ r\ x\ n\ \wedge\ r'\ x\ n\ \}\}}\text{conjunction}$$

Figure 4.13: Reasoning rules for pessimistic specifications.

We show examples that use both pessimistic and optimistic specification in Section 4.6.

Reasoning rules We can define a set of reasoning rules for the pessimistic specification and the optimistic specification, respectively. For each kind of specification, we build some reasoning rules for the five basic monadic combinators (`ret`, `bind`, `tick`, `thunk`, and `forcing`) described in Section 4.3. We can then reason about our programs purely based on these reasoning rules plus a monotonicity rule.

Figure 4.13 shows the reasoning rules for pessimistic specifications. `ret x` satisfies the pessimistic specification `r` if the result `x` and the cost 0 are in the set of `r`. `bind u k` satisfies the pessimistic specification `r` if all the results of `u` can be composed with the continuation `k` such that all the final results are in the set of `r`. A `tick` satisfies the pessimistic specification `r` if `tt` (the only value of the `unit` data type in Coq) and 1 are in the set of `r`. We also need

$$\frac{r\ \mathtt{Undefined}\ 0\ \vee\ u\ [\![\ \lambda x.\ r\ (\mathtt{Thunk}\ \ \mathtt{x})\ ]\!]}{(\mathtt{thunk}\ u)\ [\![\ r\ ]\!]}\mathtt{thunk} \qquad \frac{t = \mathtt{Thunk}\ x \qquad (k\ x)\ [\![\ r\ ]\!]}{(k\ \mathtt{\$!}\ t)\ [\![\ r\ ]\!]}\mathtt{forcing}$$

$$\frac{u\ \{\!\{\ r\ \}\!\} \qquad u\ [\![\ r'\ ]\!]}{u\ [\![\ \lambda x\ n.\ r\ x\ n\ \wedge\ r'\ x\ n\ ]\!]}\text{conjunction}$$

Figure 4.14: Reasoning rules for optimistic specifications.

a monotonicity rule to relax the pessimistic specification relation and a conjunction rule to combine pessimistic specifications.

The term `thunk u` satisfies the pessimistic specification `r` if both nondeterministic branches forked off from it satisfy the relation `r`. The `forcing` rule requires that its continuation `k` satisfies the relation `r` when applied to the value contained in a `Thunk`; in the case that there is no defined value within the thunk (*i.e.,* forcing an `Undefined`), the pessimistic specification is vacuously satisfied because the computation branch fails.

Figure 4.14 shows the reasoning rules for optimistic specifications. We omit the rules for the `ret`, `bind`, and `tick` operators and the monotonicity rule here because they have the same form as those of the pessimistic specification. There are two ways to give an optimistic specification for `thunk` terms, corresponding to selecting one of the two different nondeterministic branches that forked off from the `thunk`. In the branch where the computation is skipped, we only need to show that `Undefined` and 0 are in the relation `r`. In the branch where the computation is evaluated, we show that there exists a result in the computation `u` such that wrapping it in a `Thunk` constructor satisfies the relation `r`. The `forcing` rule requires its argument to be a defined value because forcing an `Undefined` results in failure. When reasoning about a program, we need to select the proper optimistic rule at `thunk`s so that forcing an `Undefined` value never happens.

The conjunction rule for the optimistic specification is also slightly different because its premises require both a pessimistic specification and an optimistic specification.

Approximations Before showing how we use both the pessimistic and the optimistic specifications for reasoning about lazy programs, we need to answer this question: in what sense does an approximation function implement a pure function?

Recall the approximation types and pure types discussed in Section 4.4. We would like to base our specification on pure types, as this is what we normally write as functional programs. On the other hand, our implementation in the clairvoyance monad uses approximation types.

We can connect these approximation and pure types together. First observe that we can inject pure types into partial types by thunking each subterm. We call the result an *exact* approximation because it constructs an approximation which represents *exactly* the original list.

```
Definition exact : list a -> listA a.
```

We cannot go the opposite way with a function, since approximations generally contain less information than a full list. Instead, we generalize `exact` as a relation `is_approx xsA xs` between a pure list `xs` on the right and any of its approximations on the left. A notation turns it into an infix operator with syntax inspired by Haskell.

```
Definition is_approx : listA a -> list a -> Prop.
Infix "`is_approx`" := is_approx.
```

Approximations themselves are partially ordered, when the second is at least as defined as the first. We also use infix notation for this relation.

```
Definition less_defined : listA a -> listA a -> Prop.
Infix "`less_defined`" := less_defined.
```

In our running example using lists, we also simplify things by using the same type `a` as the type of elements for pure lists `list a` and list approximations `listA a`.

More generally, we want to overload the function `exact` as well as relations `is_approx` and `less_defined`, so that (1) their names can be reused for user-defined data types, (2) they are automatically lifted through the thunk type constructor `T`.

Some properties describe and relate these three relations formally. These relations must be defined and their properties must be proved for every user-defined approximation type; those propositions and their proofs (which we omit) follow a common structure, so we conjecture that they can be automated.

The `less_defined` relation is an order relation.

```
Proposition less_defined_order : Order less_defined.
```

The set of approximations for a pure value is downward closed.

```
Proposition approx_down :
  xsA `less_defined` ysA -> ysA `is_approx` xs -> xsA `is_approx` xs.
```

The list `xsA` is an approximation of `xs` if and only if `xsA` is less defined than the exact approximation of `xs`.

```
Proposition approx_exact : xsA `is_approx` xs <-> xsA `less_defined` (exact xs).
```

Exact approximations are maximal elements for the `less_defined` order.

```
Proposition exact_max : exact xs `less_defined` xsA -> exact xs = xsA.
```

A reference implementation of all the definitions shown in this section as well as proofs for the above propositions on thunks and lists can be found in our public artifact (Li et al., 2021b).

Functional correctness To say that our approximation function implements the pure function, we would like two notions of correctness: (1) a *partial correctness* notion that requires all the nondeterministic results of the approximation function to be approximations

```
Theorem appendA_correct_partial {a} :
  forall (xs ys : list a) (xsA ysA : T (listA a)),
    xsA `is_approx` xs -> ysA `is_approx` ys ->
    (appendA xsA ysA) {{ fun zsA _ => zsA `is_approx` append xs ys }}.

Theorem appendA_correct_pure {a} :
  forall (xs ys : list a) (xsA ysA : T (listA a)),
    xsA = exact xs -> ysA = exact ys ->
    (appendA xsA ysA) [[ fun zsA _ => zsA = exact (append xs ys) ]].
```

Figure 4.15: Definitions of partial functional correctness and pure functional correctness.

of the result of the pure function; and (2) a *pure correctness* notion that states the existence of a maximal approximation result that is exactly the pure function's result.

We define the partial correctness of a function using the pessimistic specification, and the pure correctness of a function using the optimistic specification. For example, the partial and pure specifications of `appendA` (Section 4.4) are shown in Fig. 4.15. Given approximations of two input lists `xs` and `ys`, `appendA` always, *i.e.,* pessimistically, yields an approximation of `append xs ys`. On the other hand, `appendA` optimistically yields exactly the list `append xs ys` when applied to the exact approximations of `xs` and `ys`. Both theorems can be proved by an induction over `xs`.

Cost specifications In this section, we show how we use both the pessimistic and the optimistic specifications for reasoning about computation costs.

Using `appendA` as our running example, we first start with a pessimistic specification. Since a pessimistic specification describes all the nondeterministic branches of a clairvoyant call-by-value evaluation, it might contain spurious branches which overapproximate call-by-need evaluation too much. Thus, it can only offer a loose upper bound for the computation cost. Nevertheless, it is useful in specifying the lower bound, while we can rely on an optimistic specification to tighten the bounds.

Taking the `appendA` function (Fig. 4.11) again as our example, we can give it a pessimistic specification as follows:

```
Fixpoint sizeX {a} (n0 : nat) (xs : T (listA a)) : nat :=
  match xs with
  | Thunk NilA => n0
  | Thunk (ConsA _ xs1) => S (sizeX n0 xs1)
  | Undefined => 0
  end.

Definition is_defined {a} (t : T a) : Prop :=
  match t with
  | Thunk _ => True
  | Undefined => False
  end.
```

Figure 4.16: Definition of sizeX and is_defined.

```
Theorem appendA_cost_interval {a} : forall (xsA ysA : T (listA a)),
    (appendA xsA ysA)
    {{ fun zsA cost => 1 <= cost <= sizeX 1 xsA }}.
```

The xsA and ysA passed to the appendA function are approximations of the pure values xs and ys.

The size of approximation lists, defined in Fig. 4.16,[13] is a function intended purely for reasoning, hence we name it sizeX, with a different suffix from implementations such as appendA. It is also parameterized by the "size" of NilA, which is 0 or 1 depending on whether its presence matters for a given specification. Here, the extra unit of time accounts for the final call to appendA which matches on NilA.

A problem with this specification is that it gives only a range of the computation costs. During the actual evaluation of the function p, the takeA function would never require more than the first n elements of appendA's resulting list, but this specification of appendA fails to reflect that. We will only be able to compute that the combined cost has a lower bound of 3 and an upper bound of (sizeX 1 xsA) $+$ n $+ 1$, while in an actual call-by-need evaluation,

[13]The code in the figure has been simplified for clarity. The definition of sizeX is actually ill-formed because the type T (listA A) is not a recursive type.

the cost would never exceed $2\mathsf{n}+1$. Note that the size of `xsA` can be bigger than `n` if it is required with a higher demand elsewhere.

To address this problem, we give an optimistic specification to `append`. A first version states that `appendA` reaches the lower bound of the interval in at least one execution.

```
Theorem appendA_whnf_cost {a} : forall (xsA ysA : T (listA a)),
    (appendA xsA ysA)
    [[ fun zsA cost => cost <= 1 ]].
```

The execution of `appendA` which satisfies that specification immediately discards the computation in the `let~`, producing only a result in weak head normal form (WHNF).

That specification could be strengthened to an equality `cost = 1`. However, it is important to remember that results `(zsA, cost)` of a clairvoyant computation are formal approximations of the behavior of a lazy program. `zsA` is an approximation of the function's result, and `cost` is an upper bound on its actual cost. Hence, only upper bounds on `cost` are meaningful in optimistic specifications, while pessimistic specifications can assert both lower and upper bounds. For that reason, we leave specifications of `cost` as inequalities even though the simple specifications in this section are technically valid with equalities. Note also that pessimistic upper bounds are quite fragile; they can be broken simply by adding spurious thunks in a program.

Optimistic specifications about a single result such as `appendA_whnf_cost` are unhelpful in most proofs, of course. A more expressive way to phrase optimistic specifications is to set an arbitrary *demand*, raising the cost accordingly.

Examining executions of `appendA` more closely, we can distinguish two phases, with separate specifications. Before reaching the end of the first list `xsA`, `appendA` computes an approximation of length `n` in time `n`, for any `n` smaller than the size of `xsA`.

```
Theorem appendA_prefix_cost {a} : forall n (xsA ysA : T (listA a)),
    1 <= n <= sizeX 0 xsA ->
```

```
  (appendA xsA ysA) [[ fun zsA cost => n = sizeX 0 (Thunk zsA) /\ cost <= n ]].
```

The natural number `n` represents an explicit demand on the output of `appendA xsA ysA`: we demand an approximation with `n` constructors `ConsA`, costing at most `n` units of time.

Another specification describes the execution of `appendA` that reaches the end of the first list, yielding the most defined result—limited only by the possible partiality of `ysA`. As a necessary condition, `xsA` must be an exact approximation—modulo the definedness of its elements. Once we reach the end of the list `xsA`, the thunk `ysA` will be forced, so we require it to be defined, using the `is_defined` predicate in Fig. 4.16. This guarantees that `zsA` will be defined past the end of `xsA`, as specified by assigning a nonzero size to `NilA` in applications of `sizeX`.[14]

```
Theorem appendA_full_cost {a} : forall (xs : list a) (xsA := exact xs) ysA,
  is_defined ysA ->
  (appendA xsA ysA) [[ fun zsA cost =>
    length xs + sizeX 1 ysA = sizeX 1 (Thunk zsA) /\ cost <= length xs + 1 ]].
```

Natural numbers are not the most precise model of demand on lists: one may also specify whether and to what extent the elements of the list in the first field of `ConsA` constructors should be evaluated. In fact, approximation types such as `listA` are the most general way to model demand. However, when list elements are not explicitly used, a natural number is enough to formalize the main aspects of complexity analysis for list operations.

4.6. Case Study: Tail Recursion

We have already demonstrated our methodology on the program described in Section 4.2. Here, we show how to apply this approach in another context: reasoning about functions written with tail recursion. Tail recursion is a common optimization technique in the context of an eager semantics. However, it can be a cause of performance degradation if not used properly under lazy evaluation.

[14] The `(xsA := exact xs)` binding in the following snippet is desugared into a local definition using `let`.

Tail recursive `take` Consider a tail recursive version of the `take` function from Section 4.2, called `take'`. The key difference is the addition of an accumulator to its parameters:

```
Fixpoint take'_ {a} (n : nat) (xs : list a) (acc : list a) : list a :=
  match n with
  | O => rev acc
  | S n' => match xs with
            | nil => rev acc
            | cons x xs' => take'_ n' xs' (x :: acc)
            end
  end.

Definition take' {a} (n : nat) (xs : list a) : list a := take'_ n xs nil.
```

Even though the list must be reversed in the base case, `take'` is better in an eager programming language because the compiler can eliminate stack allocation (Friedman and Wise, 1974).

However, the original version is better for lazy evaluation, even if we ignore the cost of `rev`. To get an intuition of why, consider the case where we only demand the WHNF of the resulting list. This variant `take'` must go over n elements of the list before it returns the accumulator. In comparison, the original `take` can immediately reveal the first element of the resulting list in any of its pattern matching branches.

Mechanized reasoning With the help of mechanized reasoning, we can better understand these functions' difference from their specifications. In the specifications below, we axiomatize the cost of `rev` used in `take'_` to have a cost of 0. The axiom would not make Coq's logic unsound because we can define such a function in the clairvoyance monad by not inserting `ticks`. We introduce this axiom so we can compare only the costs incurred by the recursive calls on `take'_` and `take`. With this set up, let's look at the pessimistic specification of `take'A_`, the version of `take'_` written in the clairvoyance monad:[15]

[15]For simplicity, we omit the functional correctness parts of the specifications in this section.

```
forall (n : nat) (xs : list a) (xsA : listA a) (acc : list a) (accA : T (listA a)),
  xsA `is_approx` xs ->  accA `is_approx` acc ->
  (take'A_ n xsA accA) {{ fun zsA cost => cost = min n (length xs) + 1 }}.
```

The pessimistic specification describes a rather precise cost on all the nondeterministic branches of `take'A_`. Furthermore, the cost is purely decided by the pure values `n` and `xs`—the fact that the cost does not depend on the actual approximations `zsA` output by `take'A_` is a sign that the function may not be making effective use of laziness.

However, to show that `take` is better than `take'`, we need to show that `take` can cost less than `take'`. What specification should we use to show that? One possibility is the pessimistic specification. If we take this approach, we can show that the cost of `takeA` (`take` in the clairvoyance monad) is upper bounded by `n` and the size of the approximation `xsA`:

```
forall (n : nat) (xs : list a) (xsA : T (listA a)),
  xsA `is_approx` xs ->
  (takeA n xsA) {{ fun zsA cost => cost <= min n (sizeX 0 xsA) + 1 }}.
```

Since approximations are always smaller than their pure values, the cost shown here is smaller than that of `take'A_`—*if* such costs indeed exist. Unfortunately, the pessimistic specification, which quantifies universally over all executions, does not guarantee the existence of an execution. In fact, `take'A` admits no execution at all if its arguments are not sufficiently defined, so it satisfies the above specification vacuously.

To show the existence of certain costs, we need an optimistic specification:

```
forall (n m : nat) (xs : list a) (xsA : T (listA a)),
  1 <= m ->  m <= min (n + 1) (sizeX 1 xsA) ->
  xsA `is_approx` xs ->
  (takeA n xsA) [[ fun zsA cost => cost = m ]].
```

The pessimistic specification of `take'A_` and the optimistic specification of `takeA` help unveil the key difference between these two: `take'A_` does not make effective use of laziness, while the cost of `takeA` scales with its demand.

List reversal On the other hand, there are functions like `rev` that do benefit from tail recursion. Consider a naive version without tail recursion:

```
Definition rev' {a} (xs : list a) : list a :=
  match xs with
  | nil => nil
  | x :: xs' => append (rev' xs') (cons x nil)
  end.
```

And the version with tail recursion:

```
Fixpoint rev_ {a} (ys : list a) (xs : list a) : list a :=
  match xs with
  | nil => ys
  | x :: xs => rev_ (x :: ys) xs
  end.

Definition rev {a} (xs : list a) : list a := rev_ nil xs.
```

One reason that the non-tail-recursive version is worse is that `append` has a non-constant cost, which leads `rev` to have a cost which grows quadratically in the length of the input list. However, even if we imagine a constant time version of `append` (*e.g.,* difference lists (Hughes, 1986), catenable lists (Okasaki, 1999)), this version would not be better than the tail-recursive one. Intuitively, this is because both versions need to traverse the entire list `xs` to return the first element of the resulting list, which is the last element of the input list. If we consider the stack usage and compiler optimizations, the tail-recursive version would be generally more efficient and does not risk causing stack overflow.

Again we can inspect these two versions of `rev` formally to better understand their difference. Like above, we axiomatize `append` to have a cost of 0 so that our analysis only considers the cost incurred by the recursive calls of `rev_` and `rev'`. This simplification makes `rev'` cost less but will only strengthen our claim that `rev'` would not beat `rev`.

First, we can show a rather specific cost with the pessimistic specification of `revA`:

```
forall (xs : list a) (xsA : T (listA a)),
  xsA `is_approx` xs ->
  (revA xsA) {{ fun zsA cost => cost = length xs + 1 }}.
```

In that specification, the cost is associated with the pure input value, and is independent of the output value, which suggests that `revA` does not make use of laziness. Indeed, this is true: we must iterate over the entire list `xs` to get the first element of the resulting list.

We can also prove that `rev'A` satisfies the following pessimistic specification:

```
forall (xs : list a) (xsA : T (listA a)),
  xsA `is_approx` xs ->
  (rev'A xsA) {{ fun zsA cost => cost = length xs + 1 }}.
```

This specification shows that the cost of `rev'A` also does not depend the approximations of `xs`, confirming our intuition that `rev'A` must also iterate over the entire list.

Left and right folds One famous example concerning laziness is the difference between `foldl` and `foldr`. While it seems that the major difference is just their directions of folding, they actually have rather different costs. For simplicity, let's only consider these operations on `lists`. The definitions of `foldl` and `foldr` are shown below:

```
Fixpoint foldl {a b} (f : b -> a -> b) (v : b) (xs : list a) : b :=
  match xs with
  | nil => v
  | cons x xs => foldl f (f v x) xs
  end.
```

```
Fixpoint foldr {a b} (v : b) (f : a -> b -> b)  (xs : list a) : b :=
  match xs with
  | nil => v
  | cons x xs => f x (foldr v f xs)
  end.
```

A formal analysis of these functions must also consider the cost of the function `f` passed into them. For simplicity, let's assume that `f` has a cost of only 1. We can then prove that the translated versions of the above two functions satisfy the following pessimistic specification:

```
(** The pessimistic specification of [foldlA]. *)
forall f (xs : list a) (xsA : T (listA a)) (v : b) (vA : T bA),
  (forall x y, (f x y) {{ fun bA cost => exists b, bA `is_approx` b /\ cost = 1 }}) ->
  xsA `is_approx` xs ->  vA `is_approx` v ->
  (foldlA f vA xsA)
    {{ fun zsA cost => cost >= length xs + 1 /\ cost <= 2 * length xs + 1 }}.

(** The pessimistic specification of [foldrA]. *)
forall f (xs : list a) (xsA : T (listA a)) (v : b) (vA : T bA),
  (forall x y, (f x y) {{ fun bA cost => cost = 1 }}) ->
  xsA `is_approx` xs ->  vA `is_approx` v ->
  (foldrA f vA xsA)
    {{ fun zsA cost => cost >= 1 /\ cost <= 2 * sizeX 0 xsA + 1 }}.
```

The pessimistic specifications suggest that `foldrA` makes better use of laziness because its cost is bounded by the length of approximation `xsA`. However, as we have discussed earlier, we need to show that there are indeed costs lower than the lower bound of `foldlA` that exists in some nondeterministic branches of `foldrA`. For that, we once again need to show the optimistic specification of `foldrA`:

```
forall f (xs : list a) (xsA : T (listA a)) (v : b) (vA : T bA) n,
  1 <= n -> n < sizeX 0 xsA ->
  xsA `is_approx` xs ->  vA `is_approx` v ->
```

```
  (forall x y, (f x y) [[ fun bA cost => cost = 1 ]]) ->
  (foldrA f vA xsA) [[ fun zsA cost => cost = 2 * n ]].

(** And a special cost exists when [xs] is fully evaluated. *)
forall f (xs : list a) (xsA : T (listA a)) (v : b) (vA : T bA),
  xsA = exact xs ->  vA `is_approx` v -> is_defined vA ->
  (forall x y, (f x y) [[ fun bA cost => cost = 1 ]]) ->
  (foldrA f vA xsA) [[ fun zsA cost => cost = 2 * length xs + 1 ]].
```

This concludes that, under lazy evaluation, `foldl` and `foldr` have the same worst-case cost, but `foldr` has a lower cost if the demand is lower.

4.7. Extension: One Embedding to Rule Them All

In this section, I generalize the embedding rules shown in Section 4.4 to a unified embedding that works for computation cost analysis of call-by-value, call-by-name, and call-by-need (clairvoyant) programs, inspired by Petricek (2012) (Section 2.4, Fig. 2.7). The unified embedding is shown in Fig. 4.17.

In Petricek (2012), a functions of type $\tau_1 \to \tau_2$ is translated to `M` $[\![\tau_1]\!] \to$ `M` $[\![\tau_2]\!]$, where `M : Type -> Type` is a monad. Here, we instead translate such a function to `P` $[\![\tau_1]\!] \to$ `M` $[\![\tau_2]\!]$, where `P` is a parameter of type `Type -> Type`. The embedding rules for terms are almost the same as the rules in Section 4.4, except that we replace `thunk` with `pack`, and `force` with `unpack`, two functions that "convert" a datatype between `P` and `M`. The definitions of these functions depend on the evaluation strategy of the original language.

Under a call-by-value semantics, `P` is an identity functor, so the type of `pack` is equivalent to `M a -> M a` and the type of `unpack` is equivalent to `a -> M a`. Our `pack` function simply does nothing in this case, while `unpack` "wraps" a pure value inside `M`. In this case, a $[\![\text{LET } x = t \text{ IN } u]\!]$ is essentially translated to `tick >> let!` x `:=` $[\![t]\!]$ `in` $[\![u]\!]$, which forces the evaluation of $[\![t]\!]$ before running $[\![u]\!]$. This is also the same as the call-by-value semantics proposed by Wadler (1992).

Original Language: λ_{FOLDR}
Embedding Language: Coq
Embedding Domain:

```
Variable P : Type -> Type.
Variable M : Type -> Type.
Context `{Monad M}.
Parameter pack : forall {a}, M a -> M (P a).
Parameter unpack : forall {a}, P a -> M a.
```

Embedding Rules:

$$
\begin{aligned}
[\![\text{LET } x \ = \ t \text{ IN } u]\!] &= \texttt{tick >> let! } x \texttt{ := pack } [\![t]\!] \texttt{ in } [\![u]\!] \\
[\![x]\!] &= \texttt{tick >> unpack } x \\
[\![\lambda x.\ t]\!] &= \texttt{ret (fun } x \texttt{ => } [\![t]\!]\texttt{)} \\
[\![t\ x]\!] &= \texttt{tick >> let! f := } [\![t]\!] \texttt{ in f } x \\
[\![\text{NIL}]\!] &= \texttt{ret NilA} \\
[\![\text{CONS } x\ y]\!] &= \texttt{ret (ConsA } x\ y\texttt{)} \\
[\![\text{FOLDR } t_l\ (\lambda y_1\ y_2.\ t_n)\ x]\!] &= \texttt{foldrA } [\![t_l]\!] \texttt{ (fun } y_1\ y_2 \texttt{ => } [\![t_n]\!]\texttt{) } x
\end{aligned}
$$

Implementation of `pack` for λ_{FOLDR} (call-by-value):

```
Definition P := id.
Definition pack {a} : M a -> M (P a) := id.
Definition unpack {a} : P a -> M a := ret.
```

Implementation of `pack` for λ_{FOLDR} (call-by-name):

```
Definition P := M.
Definition pack {a} : M a -> M (P a) := ret.
Definition unpack {a} : P a -> M a := id.
```

Implementation of `pack` for λ_{FOLDR} (clairvoyant):

```
Definition P := T.
Definition pack {A}  M a -> M (P a) := thunk.
Definition unpack {a} : P a -> M a := force.
```

Figure 4.17: A unified embedding for λ_{FOLDR} in Coq under three different calling conventions.

Under a call-by-name semantics, `P` coincides with the monad `M`. In this case, the type of `pack` is equivalent to `M a -> M (M a)` and its definition is essentially the same as `malias` under the call-by-name semantics (Fig. 2.7). Indeed, a $[\![\text{LET } x \ = \ t \text{ IN } u]\!]$ is translated to `tick >> let!` x `:= ret` $[\![t]\!]$ `in` $[\![u]\!]$, which means that the computation $[\![t]\!]$, instead of the result of $[\![t]\!]$, is passed to `let!` (*i.e.,* `bind`).

Under a call-by-need (clairvoyant) semantics, `P` is just `T` (Fig. 4.4). The embedding rules in Fig. 4.17 is just the same as those of Fig. 4.8, if we replace `pack` with its definition `thunk`, and `unpack` with its definition `force`.

CHAPTER 5

Program Adverbs and Tlön Embeddings

This chapter references previously published paper *Program Adverbs and Tlön Embeddings* (Li and Weirich, 2022a), with adjustments to the flow and terminology. All the Coq code and theorem presented in this chapter can be found in a publicly available artifact (Li and Weirich, 2022b). I am the main contributor of the paper as well as the artifact.

The name Tlön embedding is a reference to the short story *Tlön, Uqbar, Orbis Tertius* by Jorge Luis Borges. In the short story, Tlön is an imaginary world, where its parent language does not have any nouns, but only “impersonal verbs, modified by monosyllabic suffixes (or prefixes) with an adverbial value” (Borges, 1940).

5.1. Introduction

“Where to the draw the line?”

This is a question I proposed in Section 2.3. Where to draw the line between shallow and deep embeddings is a design decision that one has to deliberately consider when using mixed embeddings. Fortunately, for many effectful programs, there is a useful guideline pointed out by research based on interaction trees (Section 3.2): modeling the pure parts of the computation “shallowly” and the effectful parts “deeply”.

Indeed, many recent efforts have focused on mixed embeddings based on interaction trees or their variants. Besides the works I mentioned in Section 3.2, there are also various applications based on free monads or their variants (Capretta, 2005; Christiansen et al., 2019; Dylus et al., 2019; Ikebuchi et al., 2022; Letan et al., 2021; McBride, 2015; Nigron and Dagand, 2021; Piróg and Gibbons, 2014; Swamy et al., 2020).[16]

[16] In this dissertation, I use free monads to refer to the inductive version of free monads. The term includes variants such as freer monads (Kiselyov and Ishii, 2015). I use interaction trees to specifically refer to the coinductive variant that I talked about in Section 3.2.

$$
\begin{array}{lrcl}
\textit{literals} & b & ::= & \text{true} \mid \text{false} \\
\textit{terms} & t, u & ::= & x \mid b \mid \neg t \mid t \wedge u \mid t \vee u
\end{array}
$$

Figure 5.1: The syntax of $\mathcal{B}_{\mathsf{var}}$.

In this chapter, we further build on this idea of separating pure and effectful parts in a mixed embedding, but inspect the following question: Why use interaction trees or free monads? Interaction trees or free monads model *one* general computation pattern that is common in many languages. However, there are other computation patterns that are not captured by these structures.

Following this observation, we propose a new class of mixed embeddings called *Tlön embeddings*. Tlön embeddings model programs using structures called *program adverbs*, which are reifications of familiar type classes (*e.g.,* `Applicative`, `Selective`, `Monad`) paired with equational theories. Like free monads, these free structures can be used to combine shallowly embedded pure computation with deeply embedded computational effects. However, program adverbs provide choices in the semantics through the selection of the structure and equational theory. For example, the "statically" adverb, based on applicative functors and their free theory, models computation where control flow and data flow in the semantics are fixed. Or, by modifying the equational theory of the free applicative structure to include commutativity, we can describe computation that is "statically and in parallel".

5.2. Embeddings for Effectful Programs

In this section, we first demonstrate embedding an effectful language using the different embedding styles covered in Chapter 2. For the purpose of demonstrating program adverbs, we consider a special scenario, where our language models a simple *circuit* that reads from *unknown external devices*. We call the original language for modeling the circuit $\mathcal{B}_{\mathsf{var}}$.

The syntax of $\mathcal{B}_{\mathsf{var}}$ appears in Fig. 5.1. Semantically, we want the Boolean operators to have their usual semantics. However, $\mathcal{B}_{\mathsf{var}}$ can read from the variables that represent references to external devices and we don't want to fix those values in the semantics. Furthermore, we don't know if the values are immutable: they might change over time, or they might change after each read, *etc.* Since $\mathcal{B}_{\mathsf{var}}$ models circuits, binary operations such as $\wedge$ and $\vee$ run in parallel.

We use $[\![\cdot]\!]_S$, $[\![\cdot]\!]_D$, $[\![\cdot]\!]_M$, and $[\![\cdot]\!]_A$ to represent embedding rules for embedding a term of $\mathcal{B}_{\mathsf{var}}$ in shallow, deep, and two mixed embeddings, respectively.

To compare embeddings, we will use each to consider the following questions regarding $\mathcal{B}_{\mathsf{var}}$:

1. Is x equivalent to $x \wedge x$?
2. Is x equivalent to $x \wedge \mathrm{true}$?
3. Is $t \wedge u$ equivalent to $u \wedge t$?
4. Is the number of variable accesses always less than or equal to 2 to the power of the circuit's depth?

Because we are modeling a circuit language that uses unknown external devices, we don't want to be able to prove or disprove property (1). This property may hold or not hold depending on the situation. If the external devices are immutable, this property will be true. Otherwise, we may be able to falsify it. In contrast, we would like our embedding to give us tools to verify properties (2) and (3) because these properties should hold regardless of the properties of our external device. The former holds because on both sides of the equivalence relation we have only accessed the variable x once. The latter is due to circuits run $\wedge$ in parallel—whatever result appears in $t \wedge u$ can also appear in $u \wedge t$ and vice versa, regardless of what effects could be involved in t or u. The last property (4) is a syntactic property of the circuit.

A shallow embedding To use a shallow embedding to represent the semantics of $\mathcal{B}_{\mathsf{var}}$, we need an embedding domain/semantic domain that can represent the effects of reading from external devices—the most common way of doing this is using *monads.* But *which*

Original Language: $\mathcal{B}_{\mathsf{var}}$
Embedding Domain:

```
Definition Reader (A : Type) : Type :=
  (var -> bool) -> A.

Definition ret {A} (a : A) : Reader A :=
  fun _ => a.

Definition bind {A B} (m : Reader A)
  (k : A -> Reader B) : Reader B :=
  fun v => k (m v) v.

Definition ask (k : var) : Reader bool :=
  fun m => m k.
```

Embedding Language: Coq
Embedding Rules:

$$
\begin{aligned}
[\![\cdot]\!]_S &: \texttt{Reader bool} \\
[\![\texttt{true}]\!]_S &= \texttt{ret true} \\
[\![\texttt{false}]\!]_S &= \texttt{ret false} \\
[\![x]\!]_S &= \texttt{ask x} \\
[\![\neg t]\!]_S &= \texttt{negb <\$>}\ [\![t]\!]_S \\
[\![t \wedge u]\!]_S &= \texttt{t'<-}\ [\![t]\!]_S\texttt{; u'<-}\ [\![u]\!]_S\texttt{;} \\
&\quad\ \texttt{ret (andb t' u')} \\
[\![t \vee u]\!]_S &= \texttt{t'<-}\ [\![t]\!]_S\texttt{; u'<-}\ [\![u]\!]_S\texttt{;} \\
&\quad\ \texttt{ret (orb t' u')}
\end{aligned}
$$

Figure 5.2: A shallow embedding for $\mathcal{B}_{\mathsf{var}}$.

one? A simple option is the reader monad (Jones, 1995; Wadler, 1992). We show core definitions of a specialized reader monad in Fig. 5.2. For simplicity, we specialize the monad so that its environment has type `var -> bool`. The commonly used reader monad is more general that the type of its environment is parameterized. Of course, the reader monad is just one possible embedding domain/semantic domain, other candidates include Dijkstra monads (Swamy et al., 2013b), predicate transformer semantics (Swierstra and Baanen, 2019), *etc.*

Using the reader monad, we can prove that property (1) is true, using $\simeq_S$, the pointwise equality of functions.

More specifically, we can prove the following Coq theorem:

```
forall x, ask x ≃S x1 <- ask x; x2 <- ask x; ret (andb x1 x2)
```

We "ask" twice on the right hand side of the equivalence to model accessing variable x twice during program runtime. However, `x1` equals to `x2` in our case since nothing has changed the global store. After proving that, the theorem can be proved via a case analysis on `x1`.

However, note that our proof relies on "nothing has changed the global store," but we don't know if this is true, as we don't know anything about the characteristics of the external device. Indeed, property (1) should *not* be true if we have a device where its values change over time: the value of x might change between two variable access. This is a problem with our choice of semantic domain. By choosing the reader monad, we introduce more assumptions over the semantics of $\mathcal{B}_{\mathsf{var}}$, which results in proving a property that is not supposed to be true in the original language $\mathcal{B}_{\mathsf{var}}$. Although this is not a problem with the approach of shallow embedding—we can choose a different monad than the reader monad, the style does force us to choose a concrete semantic domain early.

Unlike property (1), property (2) is true even though we don't know anything about the external device. This is because on both sides of the equivalence relation we have only accessed the variable x once. Property (2) can be stated as follows with our shallow embedding:

```
forall x, ask x ≃S x1 <- ask x; ret (andb x1 true)
```

The proof follows from the theories of Coq's `bool` type and the `Reader` monad. However, even though this property should be true regardless of the external device, our mechanical proof still relies on the assumption that the external device is immutable—this is again because the property is stated in terms of the reader monad, which assumes that the external environment is immutable. If we change the shallow embedding to use a different semantic domain, we would need to prove this property again.

Property (3) is true and we can prove it to be true using our shallow embedding, but that is just a lucky hit. Even though we know nothing about the external device, there is an equivalence between $t \wedge u$ and $u \wedge t$ because the two operands t and u run in parallel in a circuit. A proof based on our shallow embedding would, on the other hand, be based on the wrong assumption that the external device is immutable.

We cannot state property (4) with our shallow embedding. Our shallow embedding does not retain the syntactic structure of the original program so we cannot define a function that calculates the depth of the circuit.

Original Language: $\mathcal{B}_{\mathsf{var}}$

Embedding Domain:

```
Inductive term :=
| Var (v : var)
| Lit (b : bool)
| Neg (t : term)
| And (t : term) (u : term)
| Or (t : term) (u : term).
```

Embedding Language: Coq

Embedding Rules:

$$
\begin{aligned}
[\![\cdot]\!]_D &: \texttt{term} \\
[\![\texttt{true}]\!]_D &= \texttt{Lit true} \\
[\![\texttt{false}]\!]_D &= \texttt{Lit false} \\
[\![x]\!]_D &= \texttt{Var x} \\
[\![\neg t]\!]_D &= \texttt{Neg}\ [\![t]\!]_D \\
[\![t \wedge u]\!]_D &= \texttt{And}\ [\![t]\!]_D\ [\![u]\!]_D \\
[\![t \vee u]\!]_D &= \texttt{Or}\ [\![t]\!]_D\ [\![u]\!]_D
\end{aligned}
$$

Figure 5.3: A deep embedding for $\mathcal{B}_{\mathsf{var}}$.

A deep embedding In our deep embedding, we can use the `term` data type shown in Fig. 5.3 as the embedding domain. Our embedding rules are shown in the same figure.

Without a semantic domain, we cannot prove any of the first three properties. This is actually ideal for answering question (1) since we know nothing about the external device so we should not be able to prove it (nor should we be able to prove it wrong!). However, by leaving the entire syntax tree uninterpreted we are now unable to prove property (2) or (3), either.

A way out of this quandary is to define a coarser *equivalence relation* for ASTs and use that relation in the statement of properties (2) and (3). For example, we can interpret each `term` using the reader monad (as in the shallow embedding) and use the point-wise equivalence relation for that type. The proofs are essentially the same as the above.

However, we face a similar problem with the shallow embedding: If we would like to change the semantic domain, we need to prove our properties again. This suggests that another intermediate layer between deep and shallow embeddings might be helpful.

The primary benefit we have by using the deep embedding is that we can now state and prove property (4). This is because the deep embedding gives us a representation of the program's original syntactic structure. This allows us to define the following function that counts the depth of a circuit, similar to the `no_minus` function we define in Section 2.2:

Original Language: $\mathcal{B}_{\mathsf{var}}$

Embedding Domain:

```
Inductive FreeMonad (E : Type -> Type) R :=
| Ret (r : R)
| Bind {X} (m : E X)
    (k : X -> FreeMonad E R).

Fixpoint bind {E A B} (m : FreeMonad E A)
  (k : A -> FreeMonad E B) : FreeMonad E B :=
  match m with
  | Ret r => k r
  | Bind m' k' =>
     Bind m' (fun a => bind (k' a) k) end.

Variant DataEff : Type -> Type :=
| GetData (v : var) : DataEff bool.
```

Embedding Language: Coq

Embedding Rules:

$$
\begin{aligned}
[\![\cdot]\!]_M &: \texttt{FreeMonad DataEff bool} \\
[\![\texttt{true}]\!]_M &= \texttt{Ret true} \\
[\![\texttt{false}]\!]_M &= \texttt{Ret false} \\
[\![x]\!]_M &= \texttt{Bind (GetData x) Ret} \\
[\![\neg t]\!]_M &= \texttt{negb <\$>}\ [\![t]\!]_M \\
[\![t \wedge u]\!]_M &= \texttt{t'<-}\ [\![t]\!]_M\texttt{;}\ \texttt{u'<-}\ [\![u]\!]_M\texttt{;} \\
&\quad\ \texttt{Ret (andb t' u')} \\
[\![t \vee u]\!]_M &= \texttt{t'<-}\ [\![t]\!]_M\texttt{;}\ \texttt{u'<-}\ [\![u]\!]_M\texttt{;} \\
&\quad\ \texttt{Ret (orb t' u')}
\end{aligned}
$$

Figure 5.4: A mixed embedding based on free monads.

```
Fixpoint depth (t : term) : nat :=
  match t with
   | Var _ => 0
   | Lit _ => 0
   | Neg t => depth t + 1
   | And t u => max (depth t) (depth u) + 1
   | Or t u => max (depth t) (depth u) + 1
   end.
```

Since we assume a straightforward semantics for $\mathcal{B}_{\mathsf{var}}$, the number of variable access at runtime equals to the number of variables appeared in a `term`, so we can directly prove property (4) by an induction over the `term` data type.

A mixed embedding based on free monads The core definitions of free monads are in the left column of Fig. 5.4. They are very similar to interaction trees (Fig. 3.4): the `Ret` constructor is the same as the `Ret` constructor of `itree`, and `Bind` is the same as `Vis` of `itree`. However, `FreeMonads` are *inductive* and they do not contain a constructor that represents a silent step.

For any effect type, `FreeMonad E` is a monad as demonstrated by the `Ret` constructor and `bind` function.[17] The `bind` function pattern matches its first argument `m` and, in the case of `Bind`, passes its second arguments `k` to the continuation of `m`. This "smart constructor" ensures that binds always associate to the right.

To embed $\mathcal{B}_{\mathsf{var}}$, we model reading data from external devices using the effect type `DataEff`. This datatype includes only one (abstract) effect, called `GetData`. This constructor represents a data retrieval with the variable `v : var` that returns an unknown `bool`. Similar to how the `term` data type says nothing about the semantics of $\mathcal{B}_{\mathsf{var}}$, the effect data type `DataEff` says nothing about the semantics of a data read. As a result, we say that the effects are embedded deeply in this style.

The embedding rules appear on the right side of Fig. 5.4. The translation strategy is almost the same as embedding $\mathcal{B}_{\mathsf{var}}$ using the reader monad. The only exception is in the variable case (the effectful part): here the `Bind` constructor marks the occurrence of the `GetData` effect.

In this mixed embedding, the pure parts of a $\mathcal{B}_{\mathsf{var}}$ program have been translated to a shallow semantic domain, but the effectful parts remain abstract. It turns out that this separation is useful for both questions (1) and (2).

For question (1), we cannot answer it. This is desirable since we don't know if it's true without knowing more about the external device.

We can prove that property (2) is true even though the read effect is not interpreted—this is because the property follows from the monad laws (Fig. 5.5[18]). However, we cannot prove property (3) because the commutativity law is not one of the monad laws.

Ideally, we would also like to state and prove property (4). However, the dynamic nature of free monads forbids us from statically inspecting the syntactic structure of the program.

[17] And monad laws that can be proved in Coq.

[18] The `>>=` symbol is the infix operator for `bind`.

$$
\begin{aligned}
\textsc{Left identity} &: \texttt{ret a >>= h} = \texttt{h a} \\
\textsc{Right identity} &: \texttt{m >>= ret} = \texttt{m} \\
\textsc{Associativity} &: \texttt{(m >>= g) >>= h} = \texttt{m >>= (fun x => g x >>= h)}
\end{aligned}
$$

Figure 5.5: The monad laws.

Original Language: $\mathcal{B}_{\mathsf{var}}$
Embedding Domain:

```
Inductive ReifiedApp (E : Type -> Type) R :=
  | EmbedA (e : E R)
  | Pure (r : R)
  | LiftA2 {X Y} (f : X -> Y -> R)
    (a : ReifiedApp E X) (b : ReifiedApp E Y).
```

Embedding Language: Coq
Embedding Rules:

$$
\begin{aligned}
[\![\cdot]\!]_A &: \texttt{ReifiedApp DataEff bool} \\
[\![\texttt{true}]\!]_A &= \texttt{Pure true} \\
[\![\texttt{false}]\!]_A &= \texttt{Pure false} \\
[\![x]\!]_A &= \texttt{EmbedA (GetData x)} \\
[\![\neg t]\!]_A &= \texttt{negb <\$>} \ [\![t]\!]_A \\
[\![t \wedge u]\!]_A &= \texttt{LiftA2 andb} \ [\![t]\!]_A \ [\![u]\!]_A \\
[\![t \vee u]\!]_A &= \texttt{LiftA2 orb} \ [\![t]\!]_A \ [\![u]\!]_A
\end{aligned}
$$

Figure 5.6: A mixed embedding based on reified applicative functors.

Interpreting the embedding does not help us, either, since that would not preserve the original syntactic structure.

Our success with questions (1) and (2) suggests that we have found an useful intermediate layer between shallow and deep embeddings, but our failure in stating or proving properties (3) and (4) indicates that we haven't yet found the most suitable representation for this circuit language.

Another mixed embedding based on reified applicative functors Figure 5.6 shows a mixed embedding whose embedding domain is a type that reifies the interface of *applicative functors* (Fig. 5.7). As in free monads, this datatype is parameterized by deeply embedded abstract effects. These effects, of type `E R`, are recorded by the `EmbedA` data constructor.

However, instead of constructors for `ret` and `bind`, this datatype includes constructors for `pure` and `liftA2`, the two operations that define applicative functors.[19] The `Pure` constructor

[19] Alternatively, `Applicative` can also be defined by `pure` and another operation `<*>` of type `F (A -> B) -> F A -> F B`, where `F` is an `Applicative` instance. These two definitions are equivalent, as we can derive the definition of `<*>` from `liftA2` and vice versa.

```
Class Functor (F : Type -> Type) :=
  { fmap : forall {A B}, (A -> B) -> F A -> F B }.
Class Applicative (F : Type -> Type) `{Functor F} :=
  { pure   : forall {A}, A -> F A ;
    liftA2 : forall {A B C}, (A -> B -> C) -> F A -> F B -> F C }.
Class Selective (F : Type -> Type) `{Applicative F} :=
  { selectBy  : forall {A B C}, (A -> ((B -> C) + C)) -> F A -> F B -> F C }.
Class Monad (F : Type -> Type) `{Applicative F} :=
  { ret : forall {A}, A -> F A ;
    bind : forall {A B}, F A -> (A -> F B) -> F B }.
```

Default `fmap` *definitions*

```
Definition fmap_monad {m} `{Monad m} {a b} (f : a -> b) (x : m a) : m b :=
  x >>= (fun y => ret (f y)).
Definition fmap_ap {t}`{Applicative t}{a b} (f: a -> b) (x : t a) : t b :=
  liftA2 id (pure f) x.
```

Figure 5.7: Coq type classes for functors, applicative functors, selective functors, and monads, as well as default definitions of `fmap`.

shallowly "embeds" a pure computation into the domain, and `LiftA2` "connects" two computations that potentially contain effect invocations. These constructors provide a trivial implementation of the `Applicative` type class for this datatype.

Our embedding uses a deep embedding of variable reads, using the `EmbedA` data constructor with the `DataEff` type from the previous embedding. Because, as in free monads, this effect is modeled abstractly, we cannot prove or disprove (1).

The embedding rules use the applicative interface in the datatype to translate the constants, unary and binary operators. These components are modeled shallowly (*i.e.,* as Boolean constants and operators), but the program's syntactic structure is retained by the translation. However, because of the retainment, we need an additional equivalence relation to equate semantically equivalent terms that are not syntactically equal. To prove (2), we include the right identity law of applicative functors in the equivalence (denoted by $\cong$):

$$\frac{\forall \texttt{y},\ \texttt{(fun \_ x => x) a y = f a y}}{\texttt{liftA2 f (pure a) b} \cong \texttt{b}}$$

This law is sufficient to show that (2) holds.

To model the parallelism of circuits, we could include the commutativity law in the equivalence:

```
liftA2 f a b ≅ liftA2 (flip f) b a
```

This is sufficient to show (3). Note that this is not one of the applicative functor laws. We defer showing the soundness of including this rule in the equivalence to Section 5.3.4.

This embedding also preserves enough of the syntax of the original program to prove (4). To do so, we must first calculate the depth of circuits and the number of variables under this encoding.

```
Fixpoint app_depth {E A} (t : ReifiedApp E A) : nat :=
  match t with
   | EmbedA _ => 0
   | Pure _ => 0
   | LiftA2 _ t u => 1 + max (app_depth t) (app_depth u)
   end.

Fixpoint app_numVar {E A} (t : ReifiedApp E A) : nat :=
  match t with
   | EmbedA _ => 1
   | Pure _ => 0
   | LiftA2 _ t u => (app_numVar t) + (app_numVar u)
   end.
```

Then we can formalize (4) in Coq as follows:

```
Theorem heightAndVar : forall (c : ReifiedApp DataEff bool),
    app_numVar c <= Nat.pow 2 (app_depth c).
```

The theorem is provable by an induction over `c`.

Furthermore, this embedding also allows us to reason about semantic properties that depend on syntactic structures of circuits. One example is a semantics that includes a cost model, where accessing variables is associated with a cost. Due to this, we do not want our equivalence to equate, for example, $x \wedge u \wedge t \wedge v$ and $(x \wedge u) \wedge (t \wedge v)$ because they are not equivalent in their costs when parallelization is present. Indeed, we cannot show that they are equivalent with our embedding due to the absence of associativity in our equivalence.

Tlön embeddings Just as the reader monad models *one* particular effect, free monads model *one* particular computation pattern. Unfortunately, that particular computation pattern is not suitable for our $\mathcal{B}_{\mathsf{var}}$ example, because it does not model parallel computation (*i.e.,* property (3)), nor does it capture the static data and control flows (*i.e.,* property (4)). Instead we saw that the mixed embedding in the previous subsection, based on reified applicative functors, is a better approach.

Can we generalize the key idea even further? If we go beyond $\mathcal{B}_{\mathsf{var}}$, we might need to model other computation patterns. Are there other mixed embeddings that would be suitable for these tasks? How might we derive them?

To that end, we identify a novel set of mixed embeddings that we call *Tlön embeddings*. The goal of these embeddings is to provide flexibility in our models of effectful computation. Here, we define effects as communications with external environment that are performed by some explicit operations. For example, *mutable states* are effects which can be explicitly incurred by operations such as `get` and `set`. For the same reason, we also consider I/O (with operations like `read`, `print`, *etc.*) and exceptions (with operations like `throw`, *etc.*) as effects. We define Tlön embeddings by identifying a set of *program adverbs* that specify the embedding type and equational theory used in the embedding. For example, the embedding in Fig. 5.6 is based on an adverb composed of the `ReifiedApp` type and an equational theory based on some laws of commutative applicative functors.

The flexibility that program adverbs provide can perhaps be understood by comparing them with effects: effects *do* certain actions, and program adverbs model *how* these actions are done—similar to the difference between verbs and adverbs. For example, the adverb we used in Fig. 5.6 is called "statically and in parallel", which states that there is a static dependency between different effect invocations and some of these effect invocations are executed in parallel.

In the next section, we define our set of program adverbs more precisely and discuss the reasoning principles that they provide for effectful computation.

5.3. Program Adverbs

Program Adverbs are the building blocks of Tlön embeddings. Mathematically, they are composed of two parts: a syntactic part, called the adverb data type, and a semantic part, called the adverb theory. More formally, we define program adverbs as follows:

Definition 1 (Program Adverb)**.** *A program adverb is a pair* $(D, \cong_D)$*.* D *is called the adverb data type and is parameterized by an effect* E *and a return type* R*. The* $\cong_D$ *operation is called the adverb theory of* D*. It is a binary operation that defines an equivalence relation on* $D(E, R)$ *for any* E *and* R*.*

In the rest of the chapter, we abbreviate $\cong_D$ as $\cong$ when D is clear from the context.

In Coq terms, an adverb data type `D` has the type `(Type -> Type) -> Type -> Type`. The first parameter of `Type -> Type` is the effect E and it's parameterized by its own return type; the second parameter is the return type R. The adverb theory $\cong$ is a typed binary relation. More concretely:

```
Class Adverb (D : (Type -> Type) -> Type -> Type) :=
  { Equiv {E R} : relation (D E R) ;
    equiv {E R} : Equivalence (@Equiv E R) }.
Notation "a ≅ b" := (Equiv a b).
```

where `D` is the adverb data type, `Equiv` is the adverb theory $\cong$, and `equiv` is a proof showing that `Equiv` is an equivalence relation. The datatype `relation` is defined as:

```
Definition relation (A : Type) := A -> A -> Prop.
```

```
(* Streamingly *)
Inductive ReifiedFunctor (E : Type -> Type) (R : Type) : Type :=
| EmbedF (e : E R)
| FMap {X : Type} (g : X -> R) (f : ReifiedFunctor E X).

(* Statically and StaticallyInParallel *)
Inductive ReifiedApp (E : Type -> Type) (R : Type) : Type :=
| EmbedA (e : E R)
| Pure (r : R)
| LiftA2 {X Y : Type} (f : X -> Y -> R)
         (a : ReifiedApp E X) (b : ReifiedApp E Y).

(* Conditionally *)
Inductive ReifiedSelective (E : Type -> Type) (R : Type) : Type :=
| EmbedS (e : E R)
| PureS (r : R)
| SelectBy {X Y : Type} (f : X -> ((Y -> R) + R))
           (a : ReifiedSelective E X) (b : ReifiedSelective E Y).

(* Dynamically *)
Inductive ReifiedMonad (E : Type -> Type) (R : Type) : Type :=
| EmbedM (e : E R)
| Ret (r : R)
| Bind {X : Type} (m : ReifiedMonad E X) (k : X -> ReifiedMonad E R).
```

Figure 5.8: The adverb data types

In addition to the equivalence relations, we can also define refinement relations on program adverbs. We will show in Section 5.4.3 some adverbs with refinement relations, but equivalence relations would suffice for most adverbs, so we only include them in the core definitions of adverb theories. Refinement relations can be added on demand.

This definition is overly general, so we focus our attention only on program adverbs that are *sound* according to the definition that we will develop below. Furthermore, in this chapter we will only consider adverbs defined by reifying classes of functors.

5.3.1. Adverb Data Types and Theories

The four key adverb data types, shown in Fig. 5.8, are derived from the four type classes shown in Fig. 5.7. We have already seen one before in the applicative embedding in Fig. 5.6. Other definitions follow a similar pattern: the constructors of each data type include one for

Congruence Rule

$$\textsc{Congruence} \quad : \quad \frac{\texttt{a1} \cong \texttt{a2} \qquad \texttt{b1} \cong \texttt{b2}}{\texttt{liftA2 f a1 b1} \cong \texttt{liftA2 f a2 b2}}$$

Applicative Functor Laws

$$\textsc{Left Identity} \quad : \quad \frac{\forall \texttt{y, (fun \_ x => x) a y = f a y}}{\texttt{liftA2 f (pure a) b} \cong \texttt{b}}$$

$$\textsc{Right Identity} \quad : \quad \frac{\forall \texttt{x, (fun x \_ => x) x b = f x b}}{\texttt{liftA2 f a (pure b)} \cong \texttt{a}}$$

$$\textsc{Associativity} \quad : \quad \frac{\forall \texttt{x y z, f x y z = g y z x}}{\texttt{liftA2 id (liftA2 f a b) c} \cong \texttt{liftA2 (flip id) a (liftA2 g b c)}}$$

$$\textsc{Naturality} \quad : \quad \frac{\forall \texttt{x y z, p (q x y) z = f x (g y z)}}{\texttt{liftA2 p (liftA2 q a b)} \cong \texttt{liftA2 f a . liftA2 g b}}$$

Equivalence Properties

$$\textsc{Reflexivity} \quad : \quad \frac{}{\texttt{a} \cong \texttt{a}} \qquad \textsc{Symmetry} \quad : \quad \frac{\texttt{a} \cong \texttt{b}}{\texttt{b} \cong \texttt{a}}$$

$$\textsc{Transitivity} \quad : \quad \frac{\texttt{a} \cong \texttt{b} \qquad \texttt{b} \cong \texttt{c}}{\texttt{a} \cong \texttt{c}}$$

Figure 5.9: The equivalence relation for `ReifiedApp`.

embedding effects (of type `E R`) and a constructor that reifies the interface of each method of the type class.

In addition to an adverb data type, every program adverb also comes with some theories, defined by an equivalence relation ≅. The purpose of the ≅ relation is to equate all computations that are semantically equivalent regardless of what effects are present.

For example, an adverb called `Statically` is composed of the `ReifiedApp` datatype with an equational theory based on three sorts of rules: (1) a congruence rule with respect to `LiftA2`, (2) the laws of applicative functors (McBride and Paterson, 2008), and (3) the equivalence properties (*i.e.,* reflexivity, symmetry, transitivity). We show the concrete rules in Fig. 5.9.

```
Fixpoint interpA {E I : Type -> Type} `{Applicative I} {A : Type}
  (interpE : forall A, E A -> I A) (t : ReifiedApp E A) : I A :=
  match t with
  | EmbedA e => interpE _ e
  | Pure a => pure a
  | LiftA2 f a b => liftA2 f (interpA interpE a) (interpA interpE b)
  end.
```

Figure 5.10: The interpretation from `ReifiedApp` to any instance of the `Applicative` type class.

Why do we call this adverb `Statically`? The data dependency in the `LiftA2` constructor of `ReifiedApp` shows that the data type imposes a "static" data flow and control flow on the computation: we will always need to run both parameters of type `ReifiedApp E A` and `ReifiedApp E B` to get the result of type `ReifiedApp E C`, *i.e.,* we cannot skip either computation. In addition, neither of the two parameters depends on the result of the other, which allows us to statically inspect either of them without running the other.

Remark The adverb data types and their associated theories form free structures similar to those in Capriotti and Kaposi (2014); Kiselyov and Ishii (2015); Mokhov (2019); Mokhov et al. (2019). However, one distinction is that we intentionally do not normalize the adverb data types to preserve syntactic structures. To distinguish un-normalized free structures and normalized free structures, we use the term *reified* structures to describe the former and the term free structures to exclusively describe the latter. We defer the detailed comparison and trade-offs between reified structures and free structures to Section 5.7.

5.3.2. Adverb Simulation

One important property of `ReifiedApp` is that it can be interpreted to any other instance of the `Applicative` class, as long as its embedded effects can be interpreted to that instance. We can show this via the abstract interpreter `interpA` shown in Fig. 5.10. The interpreter shows that given *any* effect `E` and *any* instance `I` of `Applicative`, as long as we can find an effect interpretation from `E A` to `I A` for any type `A`, we can interpret a `ReifiedApp E A` to an `I A` for any type `A`.

For example, we can interpret a `ReifiedApp DataEff` to a reader applicative functor[20] (Fig. 5.2) by supplying the following function to the parameter `interpE` of `interpA`:

```
Definition interpDataEff {A : Type} (e : DataEff A) : Reader A :=
  match e with GetData v => ask v end.
```

Similarly, we can interpret `ReifiedApp DataEff` to other semantic domains that are applicative functors.

Why do we care if `ReifiedApp` can be interpreted into any instance of `Applicative`? This is because different instances of `Applicative` model different effects—if we have a data structure that can be interpreted to all instances, we can develop a theory of it that can be used for reasoning about properties that are true regardless of what effects are present.

To make the relation between an adverb data type like `ReifiedApp` and a class of functors like `Applicative` more precise, we define the following *adverb simulation* relation:

Definition 2 (Adverb Simulation)**.** *Given an adverb data type D, a class of functors C, and a transformer T on all instances of C, we say that there is an adverb simulation from D to C under T, written $D \models_T C$, if we can define a function that, for any effect type E, instance F of type class C, and interpreter f from $E(A)$ to $F(A)$ for any type A, interprets a value of $D(E, A)$ to $T(F)(A)$ for any type A.*

We add some flexibility to this definition by making it parameterize over a transformer T—we do not need this extra flexibility for now, but we will see why it is useful in Section 5.3.4.

We also define an *adverb interpretation* as follows:

Definition 3 (Adverb Interpretation)**.** *Given an adverb data type D, a class of functors C, and a transformer T on all instances of C, an interpreter I that shows $D \models_T C$ is called an adverb interpretation, and we write $I \in D \models_T C$.*

[20] Every monad is also an applicative functor, so the reader monad is also a reader applicative functor.

Our `interpA` in Fig. 5.10 is an adverb interpretation. More specifically, we say that

$$\texttt{interpA} \in \texttt{ReifiedApp} \models_{\texttt{IdT}} \texttt{Applicative}$$

where the `IdT` transformer is an identity `Applicative` transformer that "does nothing". In the rest of the chapter, when we have $D \models_{\texttt{IdT}} C$ for any D and C, we abbreviate it as $D \models C$.

5.3.3. Sound Adverb Theories

To know that our adverb theory is *sound*, *i.e.,* it doesn't equate computations that are not semantically equivalent, we define the following soundness property of adverb theories:

Definition 4 (Soundness of Adverb Theories). *Given a program adverb $(D, \cong)$ and an adverb interpretation $I \in D \models_T C$, we say that the adverb theory $\cong$ is sound with respect to I if there exists a lawful equivalence relation $\equiv$ such that for all $d_1, d_2 \in D$,*

$$d_1 \cong d_2 \implies I(d_1) \equiv I(d_2).$$

Let us use `idT` for the transformer T for the moment. The equivalence relation $\equiv$ on C is lawful if they respect the congruence laws and the class laws of C. For `Applicative`, we use the common applicative functor laws regarding $\equiv$. Based on the soundness of adverb theories, we can define the following soundness property of program adverbs with respect to their adverb interpretations:

Definition 5 (Soundness of Program Adverbs). *Given a program adverb $(D, \cong)$ and an adverb interpretation $I \in D \models_T C$, we say that the adverb is sound if the $\cong$ relation is sound with respect to I.*

We can now prove that the `Statically` adverb is sound:

Theorem 2. *The `Statically` adverb (`RefieidApp`, $\cong$) is sound with respect to the adverb interpretation `interpA` $\in$ `ReifiedApp` $\models$ `Applicative`.*

Proof. By induction over the $\cong$ relation. □

5.3.4. "Statically and in Parallel"

Two adverbs can use the same data type yet differ in their theories. Let's look at a variant of the `Statically` adverb called `StaticallyInParallel`. As its name suggests, it adds parallelization to a static computation pattern.

Recall that the two computations connected by `liftA2` do not depend on each other. This suggests that an implementation of `liftA2` can choose to run them in parallel. Indeed, that observation is one of the key ideas behind Haxl (Marlow et al., 2014).

Based on this idea, we also define the `StaticallyInParallel` adverb. The adverb data type of this adverb is the same as that of `Statically`. However, its theory differs from `Statically` in the following ways: (1) it adds the commutativity rule:

```
liftA2 f a b ≅ liftA2 (flip f) b a
```

and (2) it does not include the associativity and naturality rules (Fig. 5.9).

The addition of commutativity rule states that the order that effects are invoked does not matter. Note that compared with other rules, the commutativity rule is not satisfied by every applicative functor. This might suggest that we should not add it to the theory, as it might be a theory that only holds for certain effects. Nevertheless, we can prove the soundness of the adverb theory with respect to the following adverb simulation:

$$\texttt{ReifiedApp} \models_{\texttt{PowerSet}} \texttt{Applicative}$$

The `PowerSet` transformer is *a transformer on applicative functors* and its core definitions are shown in Fig. 5.11. The key of `PowerSet` is the `liftA2PowerSet` operation. When executed, it creates two nondeterministic branches (indicated by the disjunction `\/`): on one branch, it computes `a' : I A` before `b' : I B`, and vice versa on the other branch. Intuitively, this

```
Definition PowerSet (I : Type -> Type) (A : Type) := I A -> Prop.

Definition embedPowerSet {A : Type} (a : I A) : PowerSet I A := fun r => r ≡ a.

Definition purePowerSet {A : Type} (a : A) : PowerSet I A := fun r => r ≡ pure a.

Definition liftA2PowerSet {A B C} (f : A -> B -> C)
                     (a : PowerSet I A) (b : PowerSet I B) : PowerSet I C :=
  fun r => exists a', a a' /\ exists b', b b' /\
           (liftA2 f a' b' ≡ r \/ liftA2 (flip f) b' a' ≡ r).

Definition EqPowerSet {A} : relation (PowerSet I A) :=
  fun p q => forall a, p a <-> q a.
```

Figure 5.11: The core definitions of a powerset applicative functor transformer.

is to model the nondeterministic execution order in a parallel evaluation. Many of these operations depend on $\equiv$, which is the lawful $\equiv$ relation on `I`.

Lemma 1. *If $\equiv$ is a lawful equivalence relation on* `Applicative`*,* `EqPowerSet` *is an equivalence relation on* `PowerSet I` *that satisfies congruence, left identity, right identity, and commutativity laws.*

Proof. By definition. □

Note that `EqPowerSet I` does not satisfy the associativity or naturality laws. Consider that we have `liftA2PowerSet id (liftA2PowerSet f a b) c`, for some `f`, `a`, `b`, and `c`: one of the possible evaluations in this powerset is `liftA2 id (liftA2 (flip f) b a) c`, which does not belong to the powerset of `liftA2PowerSet (flip id) a (liftA2PowerSet g b c)`, for some `g` that is equivalent to `flip f`. The case for naturality is similar. For this reason, we do not include these two rules in $\cong$. We do not know if there exists an alternative nontrivial transformer with an equivalence relation that satisfies *all* the applicative laws in addition to commutativity.

Nevertheless, we can show the following theorem with the help of Lemma 1:

Theorem 3. *The adverb is sound:* `ReifiedApp` $\models_{\texttt{PowerSet}}$ `Applicative`*.*

Proof. We can construct an interpPowerSet $\in$ ReifiedApp $\models_{\texttt{PowerSet}}$ Applicative by modifying interpA (Fig. 5.10) so that it uses embedPowerSet on the EmbedA case, purePowerSet on the Pure case, and liftA2PowerSet on the LiftA2 case. With the help of Lemma 1, we can show that for all $d_1, d_2 \in$ ReifiedApp,

$$d_1 \cong d_2 \implies \texttt{interpPowerSet}(d_1) \equiv \texttt{interpPowerSet}(d_2)$$

where $\equiv$ is EqPowerSet. $\square$

Intuitively, we can define StaticallyInParallel as an adverb because, even though with an effect running computations in different order might return different results, a language can be implemented in a parallel way such that the difference in evaluation orders is no longer observable.

The lack of associativity and naturality rules in the theory of StaticallyInParallel might initially sound limiting, but, as we have shown in the end of Section 5.2, it turns out to be desirable for applications like circuits.

5.3.5. Other Basic Adverbs

Besides Statically and StaticallyInParallel, we also identify three other basic adverbs, namely Streamingly, Conditionally, and Dynamically, defined using the adverb data types in Fig. 5.8.

Streamingly. This program adverb simulates Functor under IdT. The most simple form of stream processing computes the data directly as it is received. This is captured by the fmap interface (Fig. 5.7).

Dynamically. This adverb simulates Monad (Fig. 5.7). A monad is the most expressive and dynamic among all four classes of functors thanks to its core operation bind. Any kind of computation can happen in the second operand and we can't know it without knowing a value of type A, which we can only get by running the first operand. This program adverb is commonly used in representing many programming language for its expressiveness, but it also allows for the least amount of static reasoning.

Unlike `Statically`, this variant does not have an `InParallel` variant. This might be surprising because there are many commutative monads. However, those monads are commutative because their specific effects are commutative. We cannot define a general powerset *monad transformer* that can make any monad satisfy the commutativity law.

`Conditionally`. We use this adverb to model conditional execution. The definition of its adverb data type is shown in Fig. 5.8. It reifies the `Selective` type class (Fig. 5.7). The signature operation of `Selective` is the `selectBy` operation. Loosely, "applying" a function of type `A -> ((B -> C) + C)` to a computation of type `F A` gets you either `F (B -> C)` or `F C`. In the first case, you will need to run the computation of type `F B`. You don't *need* to run the computation of type `F B` in the second case, but you can still choose to run it.

Because we can encode conditional execution with this adverb, it is more expressive than `Statically`. However, the extra expressiveness also makes static analysis less accurate. Since we cannot know statically if the computation `F B` in `selectBy` is executed, we can only get an under-approximation (assuming that `F B` is not executed) and an over-approximation (assuming that `F B` is executed) of the effects that would happen, but not an exact set.

Even though we derive this adverb by reifying `Selective`, we do not wish to model the adverb's theory using the laws of selective functors. This is because the laws of selective functors do not distinguish them from applicative functors. Indeed, every applicative functor is also a selective functor (by running the second argument even when not required) and vice versa, so adhering to the "default" laws do not allow us to prove more properties. Therefore, we add one simple rule to the selective functor laws:

```
select (inr <$> a) b ≅ a
```

The function `select` has the type `F (A + B) -> F (A -> B) -> F B`, where `F` is an instance of `Selective`. It is equivalent to

```
selectBy (fun x => match x with
                   | inl x => inl (fun y => y x)
                   | inr x => inr x
                   end).
```

This rule forces `select` to ignore the second argument when it does not need to be run. However, we can no longer show that the `Conditionally` adverb simulates `Selective` by adding this law, because $\cong$ is no longer an under-approximation of $\equiv$. Instead, we show the following adverb simulation:

$$\texttt{ReifiedSelective} \models \texttt{Monad}$$

In this way, `Conditionally` serves as a compromise between `Statically` and `Dynamically`. Its adverb data type is more similar to `Statically` and allows for some static analysis, while its theories are more similar to `Dynamically`.

5.4. Composable Program Adverbs

From a monad instance, we can derive an applicative functor instance. From an applicative functor instance, we can derive a functor instance. We can derive a selective instance from an applicative functor and vice versa.[21] This subsumption hierarchy among classes of functors means that we can choose the most expressive abstract interface of a data type, and that choice automatically includes the less expressive interfaces.

However, although we can derive a "default" applicative functor from a monad, we don't always want to do that—*e.g.,* we may want to define a different behavior for `liftA2` than the one derived from `bind`. Indeed, Haxl is one such example, where `bind` is defined as a sequential operation and `liftA2` is parallel so that certain tasks with no data dependencies

[21] This is one special thing about selective functors: every selective functor is an applicative functor and the reverse is also true. However, separating these two classes is still useful because the automatically derived instances might not be what we want, as discussed in Mokhov et al. (2019).

can be automatically parallelized (Marlow et al., 2014). In the program adverbs terminology, the semantics of their language is composed of a "statically and in parallel" adverb and a "dynamically" adverb.

In addition, some languages may have a part that corresponds to the "statically" adverb and some extensions that correspond to "dynamically". If we only use the "dynamically" adverb to reason about programs written in this language, we lose the ability to state properties for the "statically" subset.

We need a way to compose multiple program adverbs. Therefore, in this section, we refactor program adverbs to *composable program adverbs*.

5.4.1. Uniform Treatment of Effects and Program Adverbs

Effects are commonly considered secondary to monads. This treatment of effects carries over to the approaches based on free monads and our previous implementation of program adverbs, where the effects are a parameter of adverb data types.

This approach works well when we use one fixed program adverb, but needs an update when multiple adverbs are involved. This is because, in both scenarios we mentioned earlier, our intention is not to combine program adverbs that each contain their own set of effects—we would like the composed program adverbs to share the same set of effects. One solution is requiring that we can only join program adverbs when they share the same set of effects, but that would require extra machinery.

In our work, we choose to give a uniform treatment to effects and program adverbs. Figure 5.12 shows our algebra for effects and program adverbs. The algebra includes an $\oplus$ operator which is a *disjoint union* of effects *and* adverb data types. We define an equivalence relation $\approx$ on effects and adverb data types as follows: for all A, B that are effects and adverb data types, $A \approx B$ if there exists a *bijection* between A and B. Similarly, we define an $\uplus$ operator for the disjoint union of adverb theories. We define an equivalence relation $\Leftrightarrow$ on adverb theories as follows: for any adverb data type D and adverb theories P, Q, which

$$
\begin{array}{rrl}
\textit{effects and adverb data types} & A, B, C & ::= \texttt{Effect}\ E \mid \texttt{AdverbDataType}\ D \mid A \oplus B \\
\textit{adverb theories} & P, Q, R & ::= \texttt{AdverbTheory}\ \cong_D \mid P \uplus Q
\end{array}
$$

Properties of $\oplus$

$$
\begin{array}{rlr}
\textsc{Commutativity} & : & A \oplus B \approx B \oplus A \\
\textsc{Associativity} & : & (A \oplus B) \oplus C \approx A \oplus (B \oplus C)
\end{array}
$$

Properties of $\uplus$

$$
\begin{array}{rlr}
\textsc{Commutativity} & : & P \uplus Q \Leftrightarrow Q \uplus P \\
\textsc{Associativity} & : & (P \uplus Q) \uplus R \Leftrightarrow P \uplus (Q \uplus R) \\
\textsc{Idempotence} & : & P \uplus P \Leftrightarrow P
\end{array}
$$

Figure 5.12: The algebra for effects and composable program adverbs.

are adverb theories of D, $P \Leftrightarrow Q$ if $a\ P\ b \iff a\ Q\ b$ for all $a, b \in D$, where $\iff$ is the logical symbol for "if and only if". Properties of this algebra are also shown in Fig. 5.12.

5.4.2. The Coq Implementation

All the adverb data types we have seen (Fig. 5.8) are recursive. When we compose these program adverbs, we cannot simply put these inductive types into a sum type—we need to adapt each adverb so that it recurses on the new composed adverb rather than itself. In other words, we need *extensible inductive types*. However, extensible inductive types are not directly supported by most formal reasoning systems including Coq. In fact, how to support extensible inductive types is part of an open problem known as *the expression problem* (Wadler, 1998).

In this chapter, we address the problem and implement composable adverbs in Coq using a technique presented in *Meta Theory à la Carte* (MTC) (Delaware et al., 2013). The key idea of MTC is using Church encodings of data types (d. S. Oliveira, 2009; Wadler, 1990) instead of Coq's native inductive types. We apply and extend this idea to define the two least fixpoint operators `Fix1` and `FixRel` that work on adverb data types and adverb theories, respectively. We show the definitions of these operators in Fig. 5.13.

```
Definition Alg1 (F : (Set -> Set) -> Set -> Set) (E : Set -> Set) : Type :=
  forall {A : Set}, F E A -> E A.
Definition Fix1 (F : (Set -> Set) -> Set -> Set) (A : Set) :=
  forall (E : Set -> Set), Alg1 F E -> E A.

Definition AlgRel {F : Set -> Set}
           (R : (forall (A : Set), relation (F A)) -> forall (A : Set), relation (F A))
           (K : forall (A : Set), relation (F A)) : forall (A : Set), relation (F A) :=
  fun A (a b : F A) => R K _ a b -> K _ a b.
Definition FixRel {F : Set -> Set}
           (R : (forall (A : Set), relation (F A)) -> forall (A : Set), relation (F A))
  : forall (A : Set), relation (F A) :=
  fun A (a b : F A) => forall (K : forall (A : Set), relation (F A)),
      (forall (A : Set) (a b : F A), AlgRel R K _ a b) -> K _ a b.
```

Figure 5.13: The algebra and the least fixpoint operators for effects and adverb data types (`Alg1`, `Fix1`), and for adverb theories (`AlgRel`, `FixRel`).

```
Variant Sum1 (F G : (Set -> Set) -> Set -> Set) K R :=
  Inl1 (a : F K R) | Inr1 (a : G K R).
Variant SumRel {F : Set -> Set}
        (P Q : (forall (A : Set), relation (F A)) -> forall (A : Set), relation (F A))
        (K : forall (A : Set), relation (F A)) : forall (A : Set), relation (F A) :=
| InlRel {A : Set} {a b : F A} : P K _ a b -> SumRel P Q K _ a b
| InrRel {A : Set} {a b : F A} : Q K _ a b -> SumRel P Q K _ a b.

Notation "F ⊕ G" := (Sum1 F G).
Notation "F ⊎ G" := (SumRel F G).
```

Figure 5.14: The Coq definitions for the $\oplus$ and $\uplus$ operators.

We define the disjoint union $\oplus$ by first refactoring the types of adverb data types and effects. We make both adverb data types and effects have the type `(Set -> Set) -> Set -> Set` where the first parameter is a *recursive parameter* and the second parameter is a return type. We can then define $\oplus$ simply as a sum type on `(Set -> Set) -> Set -> Set`, as shown in Fig. 5.14. Similarly, we define $\uplus$ as a sum type on

`(forall (A : Set), relation (F A)) -> forall (A : Set), relation (F A).m`

Figure 5.15 shows the definitions of composable adverb data types. Compared with the adverb data types in Fig. 5.8, a composable adverb data type replaces the effect parame-

```
Variant ReifiedPure (K : Set -> Set) (R : Set) : Set :=
| Pure (r : R).
Variant ReifiedFunctor (K : Set -> Set) (R : Set) : Set :=
| FMap {X : Set} (g : X -> R) (f : K X).
Variant ReifiedApp (K : Set -> Set) (R : Set) : Set :=
| LiftA2 {X Y : Set} (f : X -> Y -> R)(g : K X) (a : K Y).
Variant ReifiedSelective (K : Set -> Set) (R : Set) : Set :=
| SelectBy {X Y : Set} (f : X -> ((Y -> R) + R)) (a : K X) (b : K Y).
Variant ReifiedMonad (K : Set -> Set) (R : Set) : Set :=
| Bind {X : Set} (m : K X) (g : X -> K R).
```

Figure 5.15: The composable adverb data types.

ter (which is named `E`) with a recursive parameter (which is named `K`) so that it "recurses" on `K` instead of itself.

We also factor out the `Pure` constructor, a common part shared by multiple basic adverb data types, as a separate composable adverb data type called `ReifiedPure`. In this way, we avoid introducing multiple `Pure` constructors, *e.g.,* by combining `Statically` and `Conditionally`. Furthermore, we remove the `Embed` constructors in composable adverb data types. Thanks to the uniform treatment of effects and program adverbs, we can now embed effects simply by including them in `K`, so we have no need for those constructors.

As an example, we can define an "inductive type" `T : Set -> Set` that is composed of `ReifiedPure`, `ReifiedApp`, and some effect `E : (Set -> Set) -> Set -> Set` as follows:

```
Definition T := Fix1 (ReifiedPure ⊕ ReifiedApp ⊕ E).
```

The `T` data type here is equivalent to the non-composable `ReifiedApp` shown in Fig. 5.8.

Adverb interpretation can be defined as an algebra of type `Alg1 F E` (Fig. 5.13) where `F` is the adverb data type and `E` is the instance we are interpreting to. To apply this "interpretation algebra" to the composed "inductive type", we fold it over `Fix1` as follows:

```
Definition foldFix1 {E A} (alg : Alg1 F E) (f : Fix1 F A) : E A := f _ alg.
```

We define all composable adverb data types using `Set` rather than `Type` because we use the impredicative sets extension in Coq, following MTC. The consequence of this decision is that (1) certain types cannot inhabit `Set`, and (2) the extension is inconsistent with certain set of axioms such as the axiom of unique choice together with the law of excluded middle.[22] We also develop other mechanisms like the injection type classes, the induction principles following MTC. The interested readers can find them in MTC or our supplementary artifact (Li and Weirich, 2022b).

Besides MTC, there are other solutions (Forster and Stark, 2020; Kravchuk-Kirilyuk et al., 2021) that address the expression problem in theorem provers like Coq. We discuss those alternative solutions in Section 5.7.

5.4.3. Add-on Adverbs

Another benefit of making program adverbs composable is that we can now define two add-on adverbs, namely `Repeatedly` and `Nondeterministically`, which are not suitable as standalone adverbs. These two adverbs reify two classes of functors, namely `AppKleenePlus` and `FunctorPlus`, that we define ourselves. We show these classes of functors and their reifications in Fig. 5.16. `AppKleenePlus` is a subclass of `Applicative` and represents the "Kleene plus".[23] It is a "Kleene plus" rather than a "Kleene star" because no empty element is defined. `FunctorPlus` is similar to the commonly-used `Alternative` and `MonadPlus` type classes in Haskell, but contains no empty element and only requires itself to be a subclass of `Functor`. We define these type classes' reifications as add-on adverbs so that these adverbs can be composed with classes of functors at different expressive levels: *e.g.,* `Repeatedly` can be composed with `Statically` as well as `Dynamically`.

We show the adverb theories of `Repeatedly` and `Nondeterministically` in Fig. 5.17. The function `repeat a n` repeats `a` for `n` times. Functions `kplus` and `plus` are smart constructors of `KPlus` and `Plus`, respectively. Both of these two add-on adverbs are somewhat nondetermin-

[22]https://github.com/coq/coq/wiki/Impredicative-Set
[23]https://en.wikipedia.org/wiki/Kleene_star#Kleene_plus

```
Class AppKleenePlus (F : Type -> Type) `{Applicative F} :=
  { kplus {A} : F A -> F A }.
Class FunctorPlus (F : Type -> Type) `{Functor F} :=
  { plus {A} : F A -> F A -> F A }.

(* The adverb data type for Repeatedly. *)
Variant ReifiedKleenePlus (K : Set -> Set) (R : Set) : Set :=
| KPlus : K R -> ReifiedKleenePlus K R.
(* The adverb data type for Nondeterministically. *)
Variant ReifiedPlus (K : Set -> Set) (R : Set) : Set :=
| Plus : K R -> K R -> ReifiedPlus K R.
```

Figure 5.16: The adverb data types of `Nondeterministically` and `Repeatedly`.

$$
\begin{array}{rcl}
\textsc{Repeat} & : & \forall n,\ \texttt{repeat a n} \subseteq \texttt{kplus a} \\[4pt]
\textsc{Kplus} & : & \dfrac{\texttt{a} \subseteq \texttt{kplus b}}{\texttt{kplus a} \subseteq \texttt{kplus b}} \\[8pt]
\textsc{Commutativity} & : & \texttt{plus a b} \cong \texttt{plus b a} \\[4pt]
\textsc{Associativity} & : & \texttt{plus a (plus b c)} \cong \texttt{plus (plus a b) c} \\[4pt]
\textsc{Plus} & : & \dfrac{\texttt{a} \subseteq \texttt{c} \qquad \texttt{b} \subseteq \texttt{c}}{\texttt{plus a b} \subseteq \texttt{c}} \\[8pt]
\textsc{Left Plus} & : & \texttt{a} \subseteq \texttt{plus a b} \\[4pt]
\textsc{Right Plus} & : & \texttt{b} \subseteq \texttt{plus a b}
\end{array}
$$

Figure 5.17: The adverb theories for `Repeatedly` and `Nondeterministically`.

istic, so one change we make to their adverb theories is adding refinement relations ($\subseteq$) in addition to equivalence relations ($\cong$).

We show that these two adverbs are sound with respect to the following adverb simulations:

$$\texttt{ReifiedKleenePlus} \models_{\texttt{PowerSet}} \texttt{AppKleenePlus}$$

$$\texttt{ReifiedPlus} \models_{\texttt{PowerSet}} \texttt{FunctorPlus}$$

The definition of `PowerSet` data type is the same as that in Fig. 5.11, but we are using its `AppKleenePlus` transformer and `FunctorPlus` transformer instances here. The core definitions

```
(* FunctorPlus transformer. *)
Definition fmapPowerSet {A B : Type} (f : A -> B) (a : PowerSet I A) : PowerSet I B :=
  fun r => exists a', a a' /\ fmap f a' ≡ r.

Definition plusPowerSet {A : Type} (a b : PowerSet I A) : PowerSet I A :=
  fun r => a r \/ b r.

(* AppKleenePlus transformer. *)
Definition liftA2PowerSet {A B C : Type} (f : A -> B -> C)
           (a : PowerSet I A) (b : PowerSet I B) : PowerSet I C :=
  fun r => exists a' b', a a' /\ b b' /\ (liftA2 f a' b' ≡ r).

Fixpoint repeatPowerSet {A : Type} (a : PowerSet I A) (n : nat) : PowerSet I A :=
  match n with
  | 0 => a
  | S n => liftA2PowerSet (fun _ x => x) a (repeatPowerSet a n)
  end.

Definition kplusPowerSet {A : Type} (a : PowerSet I A) : PowerSet I A :=
  fun r => exists n, repeatPowerSet a n r.
```

Figure 5.18: The `FunctorPlus` transformer instance and the `AppKleenePlus` transformer instance of the `PowerSet` data type.

of these transformers are shown in Fig. 5.18. The ≡ operator stands for the lawful equivalence relation on original functor/applicative functor `I`.

5.5. Example: Haxl

In this section, we show that we can use composable adverbs to capture two different computation patterns in the same library. We also demonstrate interpreting composable adverbs to a shallow embedding in a modular way.

We illustrate these aspects via an example based on the core ideas of Haxl. Haxl is a Haskell library developed and maintained by Meta (formerly known as Facebook) that automatically parallelizes certain operations to achieve better performance (Marlow et al., 2014). As an example, suppose that we want to fetch data from a database and we have a `Fetch : Type -> Type` data type that encapsulates the fetching effect. The key insight of the Haxl library is to distinguish the operations of `Fetch`'s `Monad` instance and those of its `Applicative` instance. When we use `>>=` to bind two `Fetch`s, those data fetches are sequential;

```
Definition Update A :=  ((var -> val) -> A * nat).

Definition ret {A} (a : A) : Update A := fun map => (a, 0).
Definition bind {A B} (m : Update A) (k : A -> Update B) : Update B :=
  fun map => match m map with
             | (i, n) => match (k i map) with
                         | (r, n') => (r, n + n')
                         end
             end.
Definition liftA2 {A B C} (f : A -> B -> C) (a : Update A) (b : Update B) : Update C :=
  fun map => match (a map, b map) with
             | ((a, n1), (b, n2)) => (f a b, max n1 n2)
             end.
Definition get (v : var) : Update val := fun map => (map v, 1).
```

Figure 5.19: The `Update` datatype.

but when we use `liftA2` to bind them, those data fetches are batched and will be sent to the database together. To achieve this, it is important that the definition of `liftA2` is not equivalent to the "default" definition derived from `>>=`.

This design of Haxl poses a challenge to mixed embeddings based on free monads or any other basic adverbs discussed in Section 5.3, because we need to distinguish when `Applicative` operations are used and when `Monad` operations are used. This is exactly where composable adverbs are useful.

In this example, we assume that we already have a translation from Haxl's `Applicative` and `Monad` operations to those operations in Coq. For example, tools like `hs-to-coq` (Spector-Zabusky, 2021) can be adapted to implement the translation. In our embedding, we use the following `T` datatype to encode the Tlön embedding of a data fetching program:

```
Definition T := Fix1 (ReifiedPure ⊕ ReifiedApp ⊕ ReifiedMonad ⊕ DataEff).
```

We use `ReifiedApp` to model batched operations and the theory of `StaticallyInParallel` to model their parallel nature. We use `ReifiedMonad` to model sequential operations.

We cannot know statically how many database accesses would happen in a Haxl program, because a program can choose to do different things depending on the result of some data fetch. Therefore, we need to pick an effect interpretation for `DataEff` to reason about this property. In this example, we are assuming that the database does not change, so we interpret our Tlön embedding to a shallow embedding whose semantic domain is the update monad (Ahman and Uustalu, 2013).

The key definitions of the update monad are shown in Fig. 5.19. The update monad is essentially a combination of a reader monad and a writer monad. In our example, the "reader state" has type `var -> val` which represents an immutable key-value database we can read from. The "writer state" is a `nat`, which represents the accumulated number of database accesses. The `bind` operation propagates the key-value database and accumulates the cost.

Additionally, we define a `liftA2` operation, which only records the maximum number of database accesses in one of its branches. This is not the same as the `liftA2` operation that can be automatically derived from the monad instance of `Update`. Furthermore, this `liftA2` is commutative. Thanks to that, we can interpret `T` to the `Update` datatype without the help of a `PowerSet` transformer.

Figure 5.20 shows how we interpret composed adverbs in a modular way. First, we define a type class called `AdverbAlg` for interpretation algebras. We then define an interpretation from each individual composable adverb and effect in `T` to `Update`. Finally, the interpretation from `T` to `Update` can be automatically inferred by Coq thanks to the instance `AdverbAlgSum`. If we would like to add another effect or composable adverb to `T`, we only need to add one more instance of `AdverbAlg` and we do not need to modify any existing interpretation algebras.

Interested readers can find the full Coq implementation of the `Update` data type, the `AdverbAlg` type class and relevant instances, along with a few simple examples in our supplementary artifact (Li and Weirich, 2022b).

```
Class AdverbAlg (D : (Set -> Set) -> Set -> Set) (I : Set -> Set) :=
  { adverbAlg : Alg1 D I }.

Instance CostApp : AdverbAlg ReifiedApp Update :=
  {| adverbAlg := fun d => match d with LiftA2 f a b => liftA2 f a b end |}.
Instance CostMonad : AdverbAlg ReifiedMonad Update :=
  {| adverbAlg := fun d => match d with Bind m k => bind m k end |}.
Instance CostPure : AdverbAlg ReifiedPure Update :=
  {| adverbAlg := fun d => match d with Pure a => ret a end |}.
Instance CostData : AdverbAlg DataEff Update :=
  {| adverbAlg := fun d => match d with GetData v => get v end |}.

Instance AdverbAlgSum D1 D2 I `{AdverbAlg D1 I} `{AdverbAlg D2 I} :
  AdverbAlg (D1 ⊕ D2) I name :=
  {| adverbAlg := fun a => match a with
                         | Inl1 a => adverbAlg a
                         | Inr1 a => adverbAlg a
                         end |}.
```

Figure 5.20: Interpretation algebras that interpret composable adverbs and `DataEff` to `Update`.

5.6. Example: Revisiting the Networked Server

A common technique used in formal verification is dividing the verification into multiple layers and establishing a refinement relation between every two layers (Gu et al., 2015; Koh et al., 2019; Lorch et al., 2020; Zakowski et al., 2021). This approach offers better abstraction and modularity, as at each layer, we only need to consider certain subsets of properties.

In this example, we show the usefulness of program adverbs and Tlön embeddings in a layered approach. Specifically, we can define an *intermediate-level* specification that omits implementation details about execution order, *etc.* Since the specification is only more *non-deterministic* in its control flow, we would like our formal verification to show that an implementation refines the specification *without* interpreting effects to a shallow embedding. This is exactly where program adverbs and Tlön embeddings can help.

We demonstrate this vision above via a much simplified version of the networked server presented in Section 3.2 and Koh et al. (2019). The server communicates with multiple clients via a network interface. A client initiates a communication with the server by sending

```
newconn ::<- accept ;;
IF (not (*newconn == 0)) THEN
  newconn_rec ::=
    connection *newconn READING ;;
  conns ::++ newconn_rec
END ;;
FOR y IN conns DO
  IF (y->state == WRITING) THEN
    r ::<- write y->id *s ;;
    y->state ::= CLOSED
  END ;;
  IF (y->state == READING) THEN
    r ::<- read y->id ;;
    IF (*r == 0) THEN
      y->state ::= CLOSED
    ELSE
      s ::= *r ;;
      y->state ::= WRITING
    END
  END
END.
```

(a) The implementation `Impl` in NetImp.

```
Some
  (Or (newconn ::<- accept ;;
       IF (not (*newconn == 0)) THEN
         newconn_rec ::=
           connection *newconn READING ;;
         conns ::++ newconn_rec
       END)
       (OneOf (conns) y
         (Or (IF (y->state == WRITING) THEN
                r ::<- write y->id *s ;;
                y->state ::= CLOSED
              END)
             (IF (y->state == READING) THEN
                r ::<- read y->id ;;
                IF (*r == 0) THEN
                  y->state ::= CLOSED
                ELSE
                  s ::= *r ;;
                  y->state ::= WRITING
                END
              END))))
```

(b) The intermediate layer specification `Spec` in NetSpec.

Figure 5.21: The implementation and the intermediate layer specification of our networked server.

a request that is a number. Whenever the server receives a request, it stores the number of that request and sends back a number in its store—a client does not necessarily receive what they sent before, because the server can interleave multiple sessions.

We show that a specific implementation of such a server refines an intermediate-level specification. We also show the refinements based on Tlön embeddings with the help of adverb theories. Unlike Koh et al. (2019), we do not show that the implementation further refines a higher-level specification based on observations over a network, as that is beyond the scope of this work.

The implementation. The server is implemented using a single-process *event loop* (Pai et al., 1999). Instead of processing a request and sending back a response immediately, the server divides a session with a client into multiple steps. In each iteration of the event loop, the server advances the session of each request by one step, thus interleaving multiple sessions.

We show the main loop body of our adapted version of the networked server in Fig. 5.21a. For simplicity, we use a custom language called NetImp. NetImp supports datatypes like booleans, natural numbers, and a special record type called `connection`. It has network operations like `accept`, `read`, and `write`. All these operations return natural numbers, with 0 indicating failures. The language does not have a while loop but it has a `FOR` loop that iterates over a list. The loop variable is implemented as a pointer that points to elements in the list iteratively. We also use C-like notations (*i.e.,* `*` and `->`) for operations on pointers.

The implementation `Impl` maintains a list of connections called `conns`. Each connection in `conns` is in one of the three possible states: `READING`, `WRITING`, or `CLOSED`. At the start of each loop, the server checks if there is a new connection waiting to be established by calling the non-blocking operation `accept`. If there is, the server creates a new `connection` with the `READING` state and adds it to `conns`. The server then goes over each `connection` in `conns`: if a connection is in the `READING` state, the server tries to read from the connection and updates an internal state `s` with the recently read value; if a connection is in the `WRITING` state, the server sends the current value of its internal state `s` to the connection; once a connection enters the `CLOSED` state, it remains that state forever and the server will not do anything with it—we design the server in this way for simplicity; a more realistic server should remove closed connections from `conns`.

The specification. We show our specification `Spec` in Fig. 5.21b. `Spec` is written in a language called NetSpec. NetSpec adds a few additional commands to NetImp: `Some` is an unary operation that models the "Kleene plus"; `Or` is a binary operation that models a nondeterministic choice; `OneOf` is also a nondeterministic choice, but it does so by choosing from a list—line 8 means that we nondeterministically assign the variable `y` with one element from the list in `conns`.

Spec is more nondeterministic compared with Impl. At each iteration of the event loop, Impl always first tries to accept a connection. It then goes over the list of conns in a fixed order. Spec does not enforce order: an accept could happen immediately after another accept; we can access elements in conns in any order and some connection might get visited more often than others.

Tlön embeddings and the refinement proof. To show that Impl refines Spec, we embed both NetImp and NetSpec in Coq using the embedding domain shown in Fig. 5.22. We have already seen the first four adverbs in T. Effect NetworkEff models the effects incurred by network operations accept, read, and write. Effect MemoryEff models the effects incurred by assigning values to variables and retrieving values from them. Finally, effect FailEff models when the program crashes.

We use $[\![\cdot]\!]^{L}_{\mathsf{T}}$ to denote a language L's Tlön embedding in T. We only show how we embed Some, Or, and OneOf in Fig. 5.22. Some is simply a kplus (from Repeatedly). Or is a plus (from Nondeterministically) wrapped inside a kplus. OneOf xs y c is a bit complicated: we first use get, an effectful MemoryEff operation that retrieves a value from the reference xs in the memory, to get a list, which we also call xs and it shadows the other xs; we then fold the list nondeterministically many times using plus over xs; each operand joined by a plus is a set y v, an effectful MemoryEff operation that set the value v in the reference y in the memory, followed by the embedding of command c.

We would like to show that $[\![\mathtt{Impl}]\!]^{I}_{\mathsf{T}} \subseteq [\![\mathtt{Spec}]\!]^{S}_{\mathsf{T}}$. Recall that $\subseteq$ is the refinement relation on program adverbs (Section 5.4.3). The theorem states that the Tlön embedding of our implementation Impl in T refines the Tlön embedding of our specification Spec in T.

To show that, we first observe that Impl and Spec share some common program fragments, *e.g.,* lines 1–6 of Impl are the same as lines 2–7 of Spec. Indeed, there are three such common fragments and we name them A (lines 1–6 of Impl), B (lines 8–11 of Impl), and C (lines 12–20 of Impl), respectively. We then define three programs L1, L2, and L3 shown in Fig. 5.23. These programs represent some intermediate layers between Impl and Spec.

Original Language: NetImp, NetSpec
Embedding Language: Coq
Embedding Domain:

```
Definition T := Fix1 (ReifiedKleenePlus ⊕
                      ReifiedPlus ⊕
                      ReifiedPure ⊕
                      ReifiedMonad ⊕
                      NetworkEff ⊕
                      MemoryEff ⊕
                      FailEff).
```

Selected Embedding Rules:

$$[\![\texttt{Some c}]\!]^S_{\mathsf{T}} = \texttt{kplus } [\![\texttt{c}]\!]^S_{\mathsf{T}}$$

$$[\![\texttt{Or c1 c2}]\!]^S_{\mathsf{T}} = \texttt{kplus (plus } [\![\texttt{c1}]\!]^S_{\mathsf{T}}\ [\![\texttt{c2}]\!]^S_{\mathsf{T}}\texttt{)}$$

$$\begin{aligned}[\![\texttt{OneOf xs y c}]\!]^S_{\mathsf{T}} = {} & \texttt{get xs >>= (fun xs => kplus (foldr} \\ & \texttt{(fun v s => plus (set y v >> } [\![\texttt{c}]\!]^S_{\mathsf{T}} \texttt{) s)} \\ & \texttt{(pure tt) xs))}\end{aligned}$$

Figure 5.22: Our Tlön embedding of NetImp and NetSpec.

```
Definition L1 :=
  A ;;
  FOR y IN conns DO
    B ;;
    C ;;
  END.
```

```
Definition L2 :=
  A ;;
  OneOf (conns) y
    (B ;; C).
```

```
Definition L3 :=
  A ;;
  OneOf (conns) y
    (Or B C).
```

Figure 5.23: Program `L1` written in NetImp, and programs `L2` and `L3` written in NetSpec.

We prove the following theorem:

Theorem 4. $[\![\mathit{Impl}]\!]^I_{T} \subseteq [\![\mathit{L1}]\!]^I_{T} \subseteq [\![\mathit{L2}]\!]^S_{T} \subseteq [\![\mathit{L3}]\!]^S_{T} \subseteq [\![\mathit{Spec}]\!]^S_{T}$.

Proof. We show $[\![\texttt{Impl}]\!]^I_{\mathsf{T}} \subseteq [\![\texttt{L1}]\!]^I_{\mathsf{T}}$ by associativity of `Dynamically`. Both $[\![\texttt{L1}]\!]^I_{\mathsf{T}} \subseteq [\![\texttt{L2}]\!]^S_{\mathsf{T}}$ and $[\![\texttt{L2}]\!]^S_{\mathsf{T}} \subseteq [\![\texttt{L3}]\!]^S_{\mathsf{T}}$ can be proven by an induction over `conns` and with the help of theories of `Dynamically`, `Repeatedly` and `Nondeterministically`. Finally, we prove $[\![\texttt{L3}]\!]^S_{\mathsf{T}} \subseteq [\![\texttt{Spec}]\!]^S_{\mathsf{T}}$ by the theories of `Dynamically`, `Repeatedly`, and `Nondeterministically`. □

Interested readers can find the full Coq implementation of NetImp, NetSpec, the Tlön embeddings of these two languages, the implementation `Impl`, the specification `Spec`, as well as the full proof of Theorem 4 in our supplementary artifact (Li and Weirich, 2022b).

5.7. Discussion

The expression problem The composable program adverbs require extensible inductive types. We implement this feature in Coq by using the Church encodings of datatypes, following the precedent work of MTC (Delaware et al., 2013). There are several consequences of using Church encodings instead of Coq's original inductive datatypes.

First, we cannot make use of Coq's language mechanisms, libraries, and plugins that make use of Coq's inductive types (*e.g.,* Coq's builtin induction principle generator, the Equations plugin (Sozeau and Mangin, 2019), the QuickChick plugin (Lampropoulos et al., 2018; Paraskevopoulou et al., 2022), *etc.*). Furthermore, the extra implementation overheads incurred by Church encodings (*e.g.,* proving an algebra is a functor, proving the induction principle using dependent types, *etc.*) can be huge. However, this situation can be helped by developing tools or plugins for supporting Church encodings.

The other consequence is that, following the practice of MTC, we use Coq's impredicative set extension. This causes two problems: (1) Certain types cannot inhabit in `Set`, and (2) our Coq development is inconsistent with certain set of axioms such as the axiom of unique choice together with the law of excluded middle, as we have discussed in Section 5.4.2.

There are alternative methods for addressing the expression problem. One option is the meta-programming approach proposed by Forster and Stark (2020). In this approach, we can define each composable adverb separately in a meta language and use a language plugin to generate a combined definition in Coq. This approach does not fully address the expression problem as extending the combined definition requires recompilation—but the amount of code that needs to be recompiled is much smaller and the generated code uses Coq's builtin inductive types. Another option that has recently been explored by Kravchuk-Kirilyuk et al.

(2021) is adding *family polymorphism* (Ernst, 2001) to theorem provers. These works are promising. Unfortunately, they either lack mature tool support or is still in development at the moment. We would like to explore these approaches in the future and composable program adverbs might provide a good application to these approaches.

Reified vs. free structures Even though the reified structures used in adverb data types are free structures, they are different from those free structures present in Capriotti and Kaposi (2014); Kiselyov and Ishii (2015); Mokhov (2019); Mokhov et al. (2019). The biggest difference between reified structures and these free structures are the parameters they recurse on: all the reified structures recurse on both their computational parameters, while each free structure only recurses on one of them.[24] For example, comparing `FreeMonad` in Fig. 5.4 and `ReifiedMonad` in Fig. 5.8: `FreeMonad` only recurses on the parameter `k` of `Bind`, while `ReifiedMonad` recurses on both parameters `m` and `k`. This means that a free structure does not just reify a class of functors, it also converts the reification to a left- or right-associative normal form.

One advantage of the normal forms in free structure definitions is that the type class laws can be automatically derived from definitional equality (with the help of the axiom of functional extensionality). However, this conversion would eliminate some differences in the syntax. Taking `ReifiedApp` as an example, normalizing it would result in a "list" rather than a "binary tree", making analyzing the depth of the tree impossible. Preserving the original tree structure of `StaticallyInParallel` also plays a crucial role in our examples shown in Section 5.2 and 5.5.

Another note is that the commonly used definition of free monads in Haskell cannot be encoded in Coq because it is not strictly positive (Dylus et al., 2019). The common ways to work around this problem are: (1) using containers (Dylus et al., 2019), or (2) using their *free* variants (Kiselyov and Ishii, 2015; McBride, 2015; Swamy et al., 2020; Xia et al., 2020).

[24] With the exception of reified/free functors, since each of them has only one computational parameters to be recursed on.

CHAPTER 6

Related work

6.1. Embeddings

Monadic embeddings Moggi (1991) uses monads to describe computational effects and defines various translations corresponding to different calling conventions. Wadler (1992) follows and describes the translations for the call-by-value and call-by-name semantics, but leaves the translation for call-by-need as an open problem. Uustalu (2002) further adds positive inductive and coinductive types to these translations.

Petricek (2012) proposes a unified monadic embedding that can be used under all three different calling conventions, generalizing Wadler (1992), by defining a function called `malias` which would be given different meanings under different semantics. We presented the embedding of Petricek (2012) in detail in Section 2.4.

Our work in Li et al. (2021a) can be extended to a unified monadic embedding that is similar to that of Petricek (2012), as discussed in Section 4.7.

Mixed embeddings It is known that there are many styles of embeddings between shallow and deep embeddings, but there is not an agreed term on describing them. In this dissertation, we use the term *mixed embeddings*, which is borrowed from Chlipala (2021). I presented an example of the style of Chlipala in Section 2.3. The style of Chlipala is also similar to higher-order abstract syntax (HOAS) (Harper et al., 1987; Pfenning and Elliott, 1988), weak HOAS (Honsell et al., 2001), and parametric HOAS (PHOAS) (Chlipala, 2008; Washburn and Weirich, 2008). I compare the style of Chlipala with HOAS and PHOAS in Section 2.3. More detailed comparison among these styles can also be found in Chlipala (2019, Section 17.2) and Chlipala (2021). Another term *deeper shallow embeddings* is proposed by Prinz et al. (2022), which shows a way of deepening any shallow embedding.

Besides mixed embeddings, there are works that study combining or transferring among different embedding styles (Fromherz et al., 2019; Svenningsson and Axelsson, 2012). For example, Svenningsson and Axelsson (2012) combine the two styles by using shallow embeddings for the interface and deep embeddings for the core language.

6.2. Reasoning about the Garden of Forking Paths

Clairvoyant evaluation Clairvoyant evaluation was first characterized by Hackett and Hutton (2019). The main inspiration for my contributions in Chapter 4, this work presented an operational semantics for laziness as an alternative to the natural semantics of Launchbury (1993), as well as a denotational cost semantics, following precursory ideas by Maraist et al. (1995). We have compared our translation with the semantics of Hackett and Hutton as well as established an equivalence relation between them in Section 4.4.

Computation Cost and Laziness There is much work on mechanically reasoning about computation costs. For example, Crary and Weirich (2000); Danielsson (2008); Hoffmann et al. (2012); Lago (2011); Rajani et al. (2021); Wang et al. (2017) study intrinsic approaches to formal cost analysis. Our work is in the extrinsic context.

On the extrinsic side, Charguéraud and Pottier (2019); Guéneau et al. (2018) use separation logic for reasoning about computation costs under call-by-value evaluation using amortized analysis. Compared to these works, our goal of reasoning about lazy pure functional programs does not require separation logic. Cost specifications could be made more modular by hiding implementation-specific constant factors and formulating costs in asymptotic terms. Works on formalizing asymptotic complexity include Cutler et al. (2020); Eberl (2021); Guéneau (2019).

Danielsson (2008); Handley et al. (2020) reason about lazy functional programs in a monadic syntax annotated with ticks. An issue in both works is that they require an explicit notion of laziness to model sharing: for example, in practice, a list that is evaluated once will not be evaluated again under lazy evaluation. To avoid a "double counting" of the cost in a thunk, a `pay : nat -> M a -> M (M a)` combinator with an explicit representation of cost must be annotated in the code[25]. This prevents both works to be fully extrinsic in reasoning about laziness. With the clairvoyance monad, thunks are either paid for or discarded immediately, so it is impossible to count the cost of a thunk twice. This enables us to translate pure lazy functions mechanically to monadic programs, and our proofs are completely extrinsic.

On the automated reasoning side, Madhavan et al. (2017) verify a purely functional subset of Scala by translating higher-order functions to first-order programs via defunctionalization. They also model memoization by encoding the cache as an expression that changes during the execution of the program.

For testing lazy functions in Haskell, Sloth is a tool that automatically generates test cases to check if a function is "unnecessarily strict" (Christiansen, 2011). This tool relies on a "less-strict" ordering of functions. One function is less strict than another when, given the same input, its result is less defined (Christiansen and Seidel, 2011).

Foner et al. (2018) develop a library that generates random demands on the output of a function and instruments inputs to record induced demand. Demands take the form of approximations whose structure is also derived from pure data types.

Haskell Although we only discuss Coq here, Haskell is also a potential target of our approach.

The `hs-to-coq` tool embeds Haskell programs in Coq using shallow embeddings (Spector-Zabusky et al., 2018). It has been used for verifying a significant portion of Haskell's `containers` library (Breitner et al., 2021) and parts of GHC (Spector-Zabusky et al., 2019), as presented in Section 3.1 However, `hs-to-coq`'s pure translation cannot be used for cost analysis so existing work using this tool has been restricted to functional correctness.

Abel et al. (2005b) and Dylus et al. (2019) respectively translate Haskell to monadic embeddings in Agda and Coq, based on the call-by-name translation by Moggi (1991). This is enough to model Haskell's partiality, but not its lazy cost semantics.

[25] The `pay` combinator is also an annotated version of `malias : M a -> M (M a)` (Petricek, 2012).

Liquid Haskell augments Haskell with refinement types (Vazou, 2016) to enable formal verification, and it has been applied to cost analysis (Handley et al., 2020). The major difference is that our work does not require an explicit notion of laziness, as discussed earlier in this section. Furthermore, Handley et al. (2020) verify Haskell programs written explicitly in the tick monad; to analyze arbitrary Haskell programs, some monadic translation is necessary.

Nondeterminism and dual logics Our optimistic and pessimistic specifications are examples of predicate transformer semantics. They date back to Dijkstra (1975), forming the basis of much work on the verification of effectful programs in type theory (Nanevski et al., 2008; Swamy et al., 2013b; Swierstra, 2009; Swierstra and Baanen, 2019). Our predicate transformer semantics are two conventional effect observations (Maillard et al., 2019) from the clairvoyance monad—a variant of the powerset monad—to the specification monads respectively for angelic and demonic nondeterminism.

The duality between pessimistic and optimistic specifications is also the duality of Hoare logic (Hoare, 1969) and reverse Hoare logic (de Vries and Koutavas, 2011; O'Hearn, 2020). Those logics use sets of states to approximate program behavior. In Hoare logic, the postcondition over-approximates the set of reachable states; in reverse Hoare logic, the postcondition under-approximates the set of reachable states. Here, we show that abstractions for angelic and demonic nondeterminism give rise, rather simply, to logics of over- and under-approximations of time consumption. The notion of approximation underlying our logics is formally defined as follows: a set of cost-value pairs A underapproximates a set of pairs B if, for every $(v, c) \in A$, there exists $(w, d) \in B$ which "costs less and is more defined", *i.e.,* such that $d \leq c$ and $v \leq w$. Thus, sets of states are ordered by inclusion in Hoare logic, whereas sets of cost-value pairs follow a more elaborate order structure in our dual logic, based on the view that those pairs themselves are approximations of the actual behavior of lazy programs.

6.3. Program Adverbs and Tlön Embeddings

Free Monads and Variants Free monads (Kiselyov and Ishii, 2015) and their variants are studied by many researchers in formal verification to reason about programs with effects. Earlier work includes the study of the *delay monad* (Capretta, 2005) and *resumption monads* (Piróg and Gibbons, 2014). More recent work includes Letan et al. (2021), where the authors use free monads to develop a modular verification framework based on effects and effect handlers called FreeSpec. Christiansen et al. (2019) develop a framework based on free monads and containers (Abbott et al., 2003) for reasoning about Haskell programs with effects. Swierstra and Baanen (2019) interpret free monads into a predicate transformer semantics that is similar to Dijkstra monads; Nigron and Dagand (2021) interprets free monads using separation logic.

On the *coinductive* side, Xia et al. (2020) develop a coinductive variant of free monads called *the interaction trees* that can be used to reason about general recursions and nonterminating programs in Coq. Koh et al. (2019) use interaction trees with VST (Appel et al., 2014) to reason about networked servers. Mansky et al. (2020) use interaction trees as a lingua franca to interface and compose higher-order separation logic in VST and a first-order verified operating system called CertiKOS (Gu et al., 2015). Zakowski et al. (2020) propose a technique called generalized parameterized coinduction for developing equational theory for reasoning about interaction trees. Zakowski et al. (2021) use interaction trees to define a modular, compositional, and executable semantics for LLVM. Yoon et al. (2022) further extend the modularity of interaction trees by extending them with layered monadic interpreters. Silver and Zdancewic (2021) connect interaction trees with Dijkstra monads (Maillard et al., 2019) for writing termination sensitive specifications based on uninterpreted effects. Lesani et al. (2022) use interaction trees to verify transactional objects. Foster et al. (2021) apply interaction trees to Isabelle/HOL to produce a verification and simulation framework for state-rich process languages, which is used by Ye et al. to give an operational semantics to RoboChart, a timed and probabilistic domain-specific language for robotics (Ye et al., 2022).

Among many variants of free monads, one particular structure closely resembles program adverbs. That is action trees defined in Swamy et al. (2020). Action trees have four constructors, `Act`, `Ret`, `Par`, and `Bind`, whose types correspond to effects, `ReifiedPure`, `ReifiedApp`, and `ReifiedMonad` in composable program adverbs, respectively, another evidence that program adverbs are general models. In contrast to our work, compositionality and extensibility of "adverbs" are not the main issue action trees try to address, so action trees are not built in a composable way. On the other hand, action trees are embedded with separation logic assertions, which are not the focus of Tlön embeddings or program adverbs.

Other Free Structures Other free structures are also explored by various works. Capriotti and Kaposi (2014) propose two variants of free applicative functors, which correspond to the left- and right-associative variants, respectively. Xia (2019) explores defining free applicative functors in Coq, and points out that the right associative variant is harder to define in Coq. Milewski (2018) discusses how to derive free monoidal functors.[26] Mokhov (2019) defines the free selective applicative functors.

In the context of mechanized reasoning, one of the main inspirations of our work is Capriotti and Kaposi (2014). They observe that the structures of free monads are not amenable to static reasoning and propose free applicative functors. Our work takes the observation further and identifies a class of program adverbs (and composable adverbs).

Programming Abstractions We are not the first to observe that monads are too dynamic for certain applications. For example, Swierstra and Duponcheel (1996) identify that a parser that has some static features cannot be defined as a monad. Inspired by their observation, Hughes (2000) proposes a new abstract interface called arrows. The relationship among arrows, applicative functors, monads are studied by Lindley et al. (2011). Willis et al. (2020) observe that monads generate dynamic structures that are hard to optimize.

[26]Monoidal functors are equivalent to applicative functors, so they also correspond to the `Statically` adverb.

They further show that, by using applicative and selective functors instead, it is possible to implement staged parser combinators that generate efficient parsers. Mokhov et al. (2020) observe that the datatype of tasks in a build system (called `Task` in their paper) can be parameterized by a class constraint to describe various kinds of build tasks. For example, a `Task Applicative` describes tasks whose dependencies are determined *statically* without running the task; and a `Task Monad` describes tasks with *dynamic* dependencies.

CHAPTER 7

Future Work

7.1. Computation Costs of Lazy Functional Data Structures

Many lazy functional data structures require subtle and intricate analysis—and they commonly require amortized analysis—to understand their computation costs. Indeed, Okasaki (1999) presents a number of such lazy functional data structures, as well as methods for analyzing their computation costs. Finger trees (Claessen, 2020; Hinze and Paterson, 2006) are another example that require complicated reasoning of their computation costs. Would the clairvoyant semantics and our dual reasoning principle help with these analyses? How do they interact with amortized analysis? And how do they interact with real-time analysis? How do they compare with classic ways of reasoning presented in Okasaki (1999)? These are interesting research questions that call for more efforts that dive deeper into the clairvoyant semantics.

7.2. Extending Tlön Embeddings to Pure Programs

In this dissertation, I show the usefulness of program adverbs and Tlön embeddings in *effectful* programs, but would they also be useful in *pure* computation?

For example, consider the purely functional FIFO queue present in Okasaki (1999, Section 5.2). The queue is implemented using two purely functional lists: a `front` list and a `back` list. Each time a new element is added into the queue (called `enqueue`), we put it into the `back` list. Each time we take an element from the queue (called `dequeue`), we take it from the `front` list. When the `front` list is empty, we reverse the `back` list and make that the new `front` list. The queue does not have constant computation costs for its `dequeue` operation. However, an amortized analysis can show that the queue has a constant computation cost for `dequeue` "on average".

We cannot express this amortized property of the queue solely based on shallow embeddings. To state this property, we need to state how many `dequeue` operations have happened. And to reason about this property, we also need to rely on the fact that the queue is used in an *ephemeral* manner, which means that every operation should only use the queue returned by its last operation and no queue should be used more than once (Driscoll et al., 1989). For these reasons, we need a syntactic representation of the queue's operations.

In this same work, Okasaki also presents a purely functional and *lazy* queue that is *persistent*, which means a queue can be used repeatedly (Okasaki, 1999, Section 6). Describing the difference between this persistent queue and the ephemeral queue also requires us to have a syntactic representation of a queue's operations.

It would be interesting future direction to explore if we can apply the methodology of program adverbs and Tlön embeddings to examples like these two queues, so that we can deeply embed operations of a certain data structure while shallow embed the others.

7.3. A Library for Composable Program Adverbs

Our work on program adverbs and Tlön embeddings have been formalized in Coq and all the Coq files are publicly available online (Li and Weirich, 2022b). However, many practical questions still need to be addressed to make our work into a Coq library that is easy to use. In particular, our work chooses to use the techniques presented in MTC (Delaware et al., 2013). As I have stated in Section 5.7, the use of Church encodings in MTC means that we cannot use Coq's language mechanisms, libraries, and plugins that make use of Coq's inductive types. Furthermore, the extra implementation overheads incurred by Church encodings can be huge. For example, the Mezzo project (Balabonski et al., 2016; Pottier and Protzenko, 2015) choose not to use the techniques presented in MTC for this reason:

> We have emphasized the modular organization of the meta-theory of Mezzo. . . The manner in which this modularity is reflected in our Coq formalization reveals pragmatic compromises. We use monolithic inductive types. Delaware et al.

> (2013) have shown how to break inductive definitions into fragments that can be modularly combined. This involves a certain notational and conceptual overhead, as well as a possible loss of flexibility, so we have not followed this route.

What can we do to reduce the implementation overhead incurred by Church encodings and the conceptual overhead of MTC? Is there a better way for Coq to support Church encodings? There are works that aim to make certain aspects of MTC easier to use (Torrini, 2016; Torrini and Schrijvers, 2015). However, there are still many questions awaiting exploration.

On the other hand, there are other ways to work around the expression problem besides MTC. I have discussed two promising directions proposed by Forster and Stark (2020) and Kravchuk-Kirilyuk et al. (2021), respectively. Alternatively, we can also consider other methodologies such as proof reuse (Ringer et al., 2019b), *etc.* It is worth exploring all alternatives and comparing them with MTC.

7.4. Other Potential Future Work

There are many other future works, such as integrating the embeddings I presented in Chapter 4 and/or Chapter 5 with `hs-to-coq`; using `hs-to-coq` and program adverbs to reason about effectful Haskell programs such as its concurrent queues (Jones et al., 1996); applying program adverbs and Tlön embeddings to verification frameworks such as VST and Iris (Jung et al., 2018); studying the connections among program adverbs, algebraic effects, object-oriented programming, and session types (Balzer and Pfenning, 2015; Zhang et al., 2020); and verifying concurrent and/or distributed systems based on Tlön embeddings.

CHAPTER 8

Conclusion

In this dissertation, I present two works on mechanized reasoning about "how" using functional programs and embeddings.

In the first work (Li, Xia, and Weirich, 2021a), we present a novel and simple shallow embedding for mechanically reasoning about costs of lazy functional programs. The embedding is based on a new model of lazy evaluation: clairvoyant call-by-value (Hackett and Hutton, 2019), which makes use of nondeterminism to avoid modeling mutable higher-order state in classic models of laziness (Launchbury, 1993).

The embedding domain of our embedding is a simple clairvoyance monad. We also propose a set of embedding rules for embedding a typed calculus to programs in this monad. Compared with the denotational semantics of Hackett and Hutton, our translation deals with typed programs, does not rely on domain theory, and accounts for the cost of every nondeterministic execution. We also develop dual logics *over-* and *under-approximations* similar to those of de Vries and Koutavas (2011); Hoare (1969); O'Hearn (2020) that enable local and modular formal reasoning of computation costs. We show the effectiveness of our approach via several small case studies.

In the second work (Li and Weirich, 2022a), we compare different styles of embeddings and how they impact mechanized reasoning about effectful programs. We find that, if used properly, mixed embeddings can combine benefits of both shallow and deep embeddings, and be effective in (1) preserving syntactic structures of original programs, (2) showing general properties that can be proved without assumptions over external environment, and (3) reasoning about properties in specialized semantic domains.

We propose *program adverbs* and *Tlön embeddings*, a class of structures and a style of mixed embeddings based on these structures, that enable us to reap these benefits. Like free monads, program adverbs embed pure computations shallowly and effects deeply (and abstractly, but can later be interpreted). However, various program adverbs correspond to alternative computation patterns, and can be composed to model programs with multiple characteristics.

Based on program adverbs, Tlön embeddings cover a wide range of programs and allow us to reason about syntactic properties, semantic properties, and general semantic properties with no assumption over external environment within the same embedding.

BIBLIOGRAPHY

Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew D. Gordon, editor, *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2620 of *Lecture Notes in Computer Science*, pages 23–38. Springer, 2003. doi: 10.1007/3-540-36576-1_2. URL https://doi.org/10.1007/3-540-36576-1_2.

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, 2005a. doi: 10.1145/1088348.1088355. URL https://doi.org/10.1145/1088348.1088355.

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, 2005b. doi: 10.1145/1088348.1088355. URL https://doi.org/10.1145/1088348.1088355.

Stephen Adams. Implementing sets efficiently in a functional language,. Research Report CSTR 92-10, University of Southampton., 1992.

Danel Ahman and Tarmo Uustalu. Update monads: Cointerpreting directed containers. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*, volume 26 of *LIPIcs*, pages 1–23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013. doi: 10.4230/LIPIcs.TYPES.2013.1. URL https://doi.org/10.4230/LIPIcs.TYPES.2013.1.

Andrew W. Appel. Coq's vibrant ecosystem for verification engineering (invited talk). In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 2–11. ACM, 2022. doi: 10.1145/3497775.3503951. URL https://doi.org/10.1145/3497775.3503951.

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014. ISBN 978-1-10-704801-0.

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 of London A: Mathematical, Physical and Engineering Sciences*, 375(2104), 2017. ISSN 1364-503X. doi: 10.1098/rsta.2016.0331.

URL http://rsta.royalsocietypublishing.org/content/375/2104/20160331.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, pages 50–65, 2005. doi: 10.1007/11541868_4. URL https://doi.org/10.1007/11541868_4.

Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 3–15. ACM, 2008. doi: 10.1145/1328438.1328443. URL https://doi.org/10.1145/1328438.1328443.

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of mezzo, a permission-based programming language. *ACM Trans. Program. Lang. Syst.*, 38(4):14:1–14:94, 2016. URL http://dl.acm.org/citation.cfm?id=2837022.

Stephanie Balzer and Frank Pfenning. Objects as session-typed processes. In Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos A. Varela, editors, *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, pages 13–24. ACM, 2015. doi: 10.1145/2824815.2824817. URL https://doi.org/10.1145/2824815.2824817.

Michael Barr and Charles Wells. *Category theory for computing science*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990. ISBN 978-0-13-120486-7.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN 978-3-642-05880-6. doi: 10.1007/978-3-662-07964-5. URL https://doi.org/10.1007/978-3-662-07964-5.

Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. 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 48–62. Springer, 2006. doi: 10.1007/978-3-540-74464-1_4. URL https://doi.org/10.1007/978-3-540-74464-1_4.

Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the C language. *J. Autom. Reason.*, 43(3):263–288, 2009. doi: 10.1007/s10817-009-9148-3. URL https://doi.org/10.1007/s10817-009-9148-3.

Jorge Luis Borges. *Tlön, Uqbar, Orbis Tertius*. In Borges et al. (2007), 1940. Translated by James E. Irby. The translation first appeared in *New World Writing* No. 18, 1961.

Jorge Luis Borges. *The Garden of Forking Paths*. In Borges et al. (2007), 1941. Translated by Donald A. Yates. The translation first appeared in *Michigan Alumnus Quarterly Review*, 1958.

Jorge Luis Borges, Donald A. Yates, James E. Irby, and Willian Gibson. *Labyrinths: Selected Stories & Other Writings*. New Directions, 2007. First published in 1962.

Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute, editors, *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992.

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). *Proc. ACM Program. Lang.*, 2(ICFP):89:1–89:16, 2018. doi: 10.1145/3236784. URL https://doi.org/10.1145/3236784.

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, Joshua Cohen, and Stephanie Weirich. Ready, `Set`, verify! applying `hs-to-coq` to real-world Haskell code. *Journal of Functional Programming*, 31:e5, 2021. doi: 10.1017/S0956796820000283.

Rod M. Burstall, David B. MacQueen, and Donald Sannella. HOPE: an experimental applicative language. In *Proceedings of the 1980 LISP Conference, Stanford, California, USA, August 25-27, 1980*, pages 136–143. ACM, 1980. doi: 10.1145/800087.802799. URL https://doi.org/10.1145/800087.802799.

Venanzio Capretta. General recursion via coinductive types. *Log. Methods Comput. Sci.*, 1(2), 2005. doi: 10.2168/LMCS-1(2:1)2005. URL https://doi.org/10.2168/LMCS-1(2:1)2005.

Paolo Capriotti and Ambrus Kaposi. Free applicative functors. In Paul Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 2–30, 2014. doi: 10.4204/EPTCS.153.2. URL https://doi.org/10.4204/EPTCS.153.2.

Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *J. Autom.*

Reason., 62(3):331–365, 2019. doi: 10.1007/s10817-017-9431-7. URL https://doi.org/10.1007/s10817-017-9431-7.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 18–37. ACM, 2015. doi: 10.1145/2815400.2815402. URL https://doi.org/10.1145/2815400.2815402.

Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008. doi: 10.1145/1411204.1411226. URL https://doi.org/10.1145/1411204.1411226.

Adam Chlipala. *Certified Programming with Dependent Types.* Electronic textbook, April 2019. URL http://adam.chlipala.net/cpdt/. Licensed under a Creative Commons Attribution-Noncommercial-NoDerivative Works 3.0 Unported License.

Adam Chlipala. Skipping the binder bureaucracy with mixed embeddings in a semantics course (functional pearl). *Proc. ACM Program. Lang.*, 5(ICFP):1–28, 2021. doi: 10.1145/3473599. URL https://doi.org/10.1145/3473599.

Adam Chlipala. *Formal Reasoning About Programs.* Electronic textbook, April 2022. URL http://adam.chlipala.net/frap/. Licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

Jan Christiansen. Sloth - A tool for checking minimal-strictness. In Ricardo Rocha and John Launchbury, editors, *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*, volume 6539 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2011. doi: 10.1007/978-3-642-18378-2_14. URL https://doi.org/10.1007/978-3-642-18378-2_14.

Jan Christiansen and Daniel Seidel. Minimally strict polymorphic functions. In Peter Schneider-Kamp and Michael Hanus, editors, *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 53–64. ACM, 2011. doi: 10.1145/2003476.2003487. URL https://doi.org/10.1145/2003476.2003487.

Jan Christiansen, Sandra Dylus, and Niels Bunkenburg. Verifying effectful Haskell programs in Coq. In Richard A. Eisenberg, editor, *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, pages 125–138. ACM, 2019. doi: 10.1145/3331545.3342592. URL https://doi.org/10.1145/3331545.3342592.

Koen Claessen. Finger trees explained anew, and slightly simplified (functional pearl). In Tom Schrijvers, editor, *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020*, pages 31–38. ACM, 2020. doi: 10.1145/3406088.3409026. URL https://doi.org/10.1145/3406088.3409026.

Karl Crary and Stephanie Weirich. Resource bound certification. In Mark N. Wegman and Thomas W. Reps, editors, *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 184–198. ACM, 2000. doi: 10.1145/325694.325716. URL https://doi.org/10.1145/325694.325716.

Joseph W Cutler, Daniel R Licata, and Norman Danner. Denotational recurrence extraction for amortized analysis. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29, 2020.

Bruno C. d. S. Oliveira. Modular visitor components. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 269–293. Springer, 2009. doi: 10.1007/978-3-642-03013-0_13. URL https://doi.org/10.1007/978-3-642-03013-0_13.

Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 133–144. ACM, 2008. doi: 10.1145/1328438.1328457. URL https://doi.org/10.1145/1328438.1328457.

Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 206–217. ACM, 2006. doi: 10.1145/1111037.1111056. URL https://doi.org/10.1145/1111037.1111056.

Edsko de Vries and Vasileios Koutavas. Reverse Hoare logic. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*, volume 7041 of *Lecture Notes in Computer Science*, pages 155–171. Springer, 2011. doi: 10.1007/978-3-642-24690-6_12. URL https://doi.org/10.1007/978-3-642-24690-6_12.

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. 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 207–218. ACM, 2013. doi: 10.1145/2429069.2429094. URL https://doi.org/10.1145/2429069.2429094.

Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989. doi: 10.1016/0022-0000(89)90034-2. URL https://doi.org/10.1016/0022-0000(89)90034-2.

Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying Haskell programs by combining testing, model checking and interactive theorem proving. *Inf. Softw. Technol.*, 46 (15):1011–1025, 2004. doi: 10.1016/j.infsof.2004.07.002. URL https://doi.org/10.1016/j.infsof.2004.07.002.

Sandra Dylus, Jan Christiansen, and Finn Teegen. One monad to prove them all. *Art Sci. Eng. Program.*, 3(3):8, 2019. doi: 10.22152/programming-journal.org/2019/3/8. URL https://doi.org/10.22152/programming-journal.org/2019/3/8.

Manuel Eberl. *Asymptotic Reasoning in a Proof Assistant*. PhD thesis, Technical University of Munich, Munich, Germany, 2021.

Richard A. Eisenberg. System FC, as implemented in GHC, 2020. URL https://github.com/ghc/ghc/blob/master/docs/core-spec/core-spec.pdf.

Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, 2001. doi: 10.1007/3-540-45337-7_17. URL https://doi.org/10.1007/3-540-45337-7_17.

Kenneth Foner, Hengchu Zhang, and Leonidas Lampropoulos. Keep your laziness in check. *Proc. ACM Program. Lang.*, 2(ICFP):102:1–102:30, 2018. doi: 10.1145/3236797. URL https://doi.org/10.1145/3236797.

Yannick Forster and Kathrin Stark. Coq à la carte: a practical approach to modular syntax with binders. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 186–200. ACM, 2020. doi: 10.1145/3372885.3373817. URL https://doi.org/10.1145/3372885.3373817.

Simon Foster, Chung-Kil Hur, and Jim Woodcock. Formally verified simulations of state-rich processes using interaction trees in isabelle/hol. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.CONCUR.2021.20. URL https://doi.org/10.4230/LIPIcs.CONCUR.2021.20.

Daniel P. Friedman and David S. Wise. Unwinding structured recursions into iterations. Technical Report TR19, Indiana University, 1974. URL https://help.luddy.indiana.edu/techreports/TRNNN.cgi?trnum=TR19.

Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. *Proc. ACM Program. Lang.*, 3(POPL):63:1–63:30, 2019. doi: 10.1145/3290376. URL https://doi.org/10.1145/3290376.

Jeremy Gibbons and Ralf Hinze. Just `do` it: simple monadic equational reasoning. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 2–14. ACM, 2011. doi: 10.1145/2034773.2034777. URL https://doi.org/10.1145/2034773.2034777.

Gabriel Gonzalez. Equational reasoning, December 2013. URL http://www.haskellforall.com/2013/12/equational-reasoning.html. Blog post.

Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 595–608. ACM, 2015. doi: 10.1145/2676726.2676975. URL https://doi.org/10.1145/2676726.2676975.

Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. Building certified concurrent OS kernels. *Commun. ACM*, 62(10):89–99, 2019. doi: 10.1145/3356903. URL https://doi.org/10.1145/3356903.

Armaël Guéneau. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs. (Vérification mécanisée de la correction et complexité asymptotique de programmes)*. PhD thesis, Inria, Paris, France, 2019. URL https://tel.archives-ouvertes.fr/tel-02437532.

Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 533–560. Springer, 2018. doi: 10.1007/978-3-319-89884-1_19. URL https://doi.org/10.1007/978-3-319-89884-1_19.

Jennifer Hackett and Graham Hutton. Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.*, 3(ICFP):114:1–114:23, 2019. doi: 10.1145/3341718. URL https://doi.org/10.1145/3341718.

Thomas Hallgren. Haskell tools from the programatica project. In Johan Jeuring, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*, pages 103–106. ACM, 2003. doi: 10.1145/871895.871907. URL https://doi.org/10.1145/871895.871907.

Martin A. T. Handley, Niki Vazou, and Graham Hutton. Liquidate your assets: reasoning about resource usage in Liquid Haskell. *Proc. ACM Program. Lang.*, 4(POPL):24:1–24:27, 2020. doi: 10.1145/3371092. URL https://doi.org/10.1145/3371092.

Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*, pages 194–204. IEEE Computer Society, 1987.

Peter Henderson and James H. Morris, Jr. A lazy evaluator. In Susan L. Graham, Robert M. Graham, Michael A. Harrison, William I. Grosky, and Jeffrey D. Ullman, editors, *Conference Record of the Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, USA, January 1976*, pages 95–103. ACM Press, 1976. doi: 10.1145/800168.811543. URL https://doi.org/10.1145/800168.811543.

Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006. doi: 10.1017/S0956796805005769. URL https://doi.org/10.1017/S0956796805005769.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, 2012. doi: 10.1145/2362389.2362393. URL https://doi.org/10.1145/2362389.2362393.

Furio Honsell, Marino Miculan, and Ivan Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, volume 2076 of *Lecture Notes in Computer Science*, pages 963–978. Springer, 2001. doi: 10.1007/3-540-48224-5_78. URL https://doi.org/10.1007/3-540-48224-5_78.

John Hughes. A novel representation of lists and its application to the function "reverse". *Inf. Process. Lett.*, 22(3):141–144, 1986. doi: 10.1016/0020-0190(86)90059-1. URL https://doi.org/10.1016/0020-0190(86)90059-1.

John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989. doi: 10.1093/comjnl/32.2.98. URL https://doi.org/10.1093/comjnl/32.2.98.

John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000. doi: 10.1016/S0167-6423(99)00023-4. URL https://doi.org/10.1016/S0167-6423(99)00023-4.

Mirai Ikebuchi, Andres Erbsen, and Adam Chlipala. Certifying derivation of state machines from coroutines. *Proc. ACM Program. Lang.*, 6(POPL):1–31, 2022. doi: 10.1145/3498685. URL https://doi.org/10.1145/3498685.

Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer, 1995. doi: 10.1007/3-540-59451-5_4. URL https://doi.org/10.1007/3-540-59451-5_4.

Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjørn Finne. Concurrent Haskell. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 295–308. ACM Press, 1996. doi: 10.1145/237721.237794. URL https://doi.org/10.1145/237721.237794.

Mark B. Josephs. The semantics of lazy functional languages. *Theor. Comput. Sci.*, 68(1):105–111, 1989. doi: 10.1016/0304-3975(89)90122-9. URL https://doi.org/10.1016/0304-3975(89)90122-9.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi: 10.1017/S0956796818000151. URL https://doi.org/10.1017/S0956796818000151.

Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105, 2015. doi: 10.1145/2804302.2804319. URL http://doi.acm.org/10.1145/2804302.2804319.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From C to interaction trees: specifying, verifying, and testing a networked server. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 234–248. ACM, 2019. doi: 10.1145/3293880.3294106. URL https://doi.org/10.1145/3293880.3294106.

Anastasiya Kravchuk-Kirilyuk, Yizhou Zhang, and Nada Amin. Family polymorphism for proof extensibility. In *Proceedings of the 27th International Conference on Types for Proofs and Programs, TYPES 2021, June 14–18, 2021*, 2021. URL https://types21.liacs.nl/download/family-polymorphism-for-proof-extensibility/.

Ugo Dal Lago. A short introduction to implicit computational complexity. In Nick Bezhanishvili and Valentin Goranko, editors, *Lectures on Logic and Computation - ESSLLI 2010 Copenhagen, Denmark, August 2010, ESSLLI 2011, Ljubljana, Slovenia, August 2011, Selected Lecture Notes*, volume 7388 of *Lecture Notes in Computer Science*, pages 89–109. Springer, 2011. doi: 10.1007/978-3-642-31485-8_3. URL https://doi.org/10.1007/978-3-642-31485-8_3.

Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, May 2021. Version 1.2. https://softwarefoundations.cis.upenn.edu/qc-1.2/.

Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL):45:1–45:30, 2018. doi: 10.1145/3158133. URL https://doi.org/10.1145/3158133.

John Launchbury. A natural semantics for lazy evaluation. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 144–154. ACM Press, 1993. doi: 10.1145/158511.158618. URL https://doi.org/10.1145/158511.158618.

John Launchbury and Simon L. Peyton Jones. State in Haskell. *LISP Symb. Comput.*, 8 (4):293–341, 1995.

Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. C4: verified transactional objects. *Proc. ACM Program. Lang.*, 6(OOPSLA):1–31, 2022. doi: 10.1145/3527324. URL https://doi.org/10.1145/3527324.

Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effects handlers. *Formal Aspects Comput.*, 33

(1):127–150, 2021. doi: 10.1007/s00165-020-00523-2. URL https://doi.org/10.1007/s00165-020-00523-2.

Yao Li and Stephanie Weirich. Program adverbs and Tlön embeddings. *Proc. ACM Program. Lang.*, 6(ICFP), 2022a.

Yao Li and Stephanie Weirich. Program adverbs and tlön embeddings (artifact), June 2022b. URL https://doi.org/10.5281/zenodo.6678339.

Yao Li, Li-yao Xia, and Stephanie Weirich. Reasoning about the garden of forking paths. *Proc. ACM Program. Lang.*, 5(ICFP):1–28, 2021a. doi: 10.1145/3473585. URL https://doi.org/10.1145/3473585.

Yao Li, Li-yao Xia, and Stephanie Weirich. Reasoning about the garden of forking paths (artifact), August 2021b. URL https://doi.org/10.5281/zenodo.5154097.

Yishuai Li, Benjamin C. Pierce, and Steve Zdancewic. Model-based testing of networked applications. In Cristian Cadar and Xiangyu Zhang, editors, *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, pages 529–539. ACM, 2021c. doi: 10.1145/3460319.3464798. URL https://doi.org/10.1145/3460319.3464798.

Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 333–343. ACM Press, 1995. doi: 10.1145/199448.199528. URL https://doi.org/10.1145/199448.199528.

Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electron. Notes Theor. Comput. Sci.*, 229(5):97–117, 2011. doi: 10.1016/j.entcs.2011.02.018. URL https://doi.org/10.1016/j.entcs.2011.02.018.

Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: low-effort verification of high-performance concurrent programs. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 197–210. ACM, 2020. doi: 10.1145/3385412.3385971. URL https://doi.org/10.1145/3385412.3385971.

Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based resource verification for higher-order functions with memoization. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 330–343. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3009874.

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP): 104:1–104:29, 2019. doi: 10.1145/3341708. URL https://doi.org/10.1145/3341708.

William Mansky, Wolf Honoré, and Andrew W. Appel. Connecting higher-order separation logic to a first-order outside world. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 428–455. Springer, 2020. doi: 10.1007/978-3-030-44914-8_16. URL https://doi.org/10.1007/978-3-030-44914-8_16.

John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. In Stephen D. Brookes, Michael G. Main, Austin Melton, and Michael W. Mislove, editors, *Eleventh Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 1995, Tulane University, New Orleans, LA, USA, March 29 - April 1, 1995*, volume 1 of *Electronic Notes in Theoretical Computer Science*, pages 370–392. Elsevier, 1995. doi: 10.1016/S1571-0661(04)00022-2. URL https://doi.org/10.1016/S1571-0661(04)00022-2.

Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. There is no fork: an abstraction for efficient, concurrent, and concise data access. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 325–337. ACM, 2014. doi: 10.1145/2628136.2628144. URL https://doi.org/10.1145/2628136.2628144.

Coq development team. *The Coq proof assistant*, 2022. URL http://coq.inria.fr. Version 8.15.1.

Conor McBride. Turing-completeness totally free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, pages 257–275, 2015. doi: 10.1007/978-3-319-19797-5_13. URL https://doi.org/10.1007/978-3-319-19797-5_13.

Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008. doi: 10.1017/S0956796807006326. URL https://doi.org/10.1017/S0956796807006326.

Bartosz Milewski. Free monoidal functors, categorically!, May 2018. URL https://bartoszmilewski.com/2018/05/16/free-monoidal-functors-categorically/. Blog post.

Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991. doi: 10.1016/0890-5401(91)90052-4. URL https://doi.org/10.1016/0890-5401(91)90052-4.

Andrey Mokhov. Implementation of selective applicative functors in Haskell, 2019. URL https://hackage.haskell.org/package/selective.

Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jerémie Dimino. Selective applicative functors. *Proc. ACM Program. Lang.*, 3(ICFP):90:1–90:29, 2019. doi: 10.1145/3341694. URL https://doi.org/10.1145/3341694.

Andrey Mokhov, Neil Mitchell, and Simon L. Peyton Jones. Build systems à la carte: Theory and practice. *J. Funct. Program.*, 30:e11, 2020. doi: 10.1017/S0956796820000088. URL https://doi.org/10.1017/S0956796820000088.

Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 43–56. ACM, 1999. doi: 10.1145/292540.292547. URL https://doi.org/10.1145/292540.292547.

John Garrett Morris. *Type Classes and Instance Chains: A Relational Approach*. PhD thesis, Portland State University, Portland, OR, USA, 2013. URL http://archives.pdx.edu/ds/psu/9917.

Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008. doi: 10.1017/S0956796808006953. URL https://doi.org/10.1017/S0956796808006953.

Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. *SIAM J. Comput.*, 2(1):33–43, 1973. doi: 10.1137/0202005. URL https://doi.org/10.1137/0202005.

Pierre Nigron and Pierre-Évariste Dagand. Reaching for the star: Tale of a monad in coq. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 29:1–29:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.ITP.2021.29. URL https://doi.org/10.4230/LIPIcs.ITP.2021.29.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-43376-7. doi: 10.1007/3-540-45949-9. URL https://doi.org/10.1007/3-540-45949-9.

Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, 2020. doi: 10.1145/3371078. URL https://doi.org/10.1145/3371078.

Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. ISBN 978-0-521-66350-2.

Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the 1999 USENIX Annual Technical Conference, June 6-11, 1999, Monterey, California, USA*, pages 199–212. USENIX, 1999. URL http://www.usenix.org/events/usenix99/full_papers/pai/pai.pdf.

Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015. doi: 10.1007/978-3-319-22102-1_22. URL https://doi.org/10.1007/978-3-319-22102-1_22.

Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. Computing correctly with inductive relations. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 966–980. ACM, 2022. doi: 10.1145/3519939.3523707. URL https://doi.org/10.1145/3519939.3523707.

Tomas Petricek. Evaluation strategies for monadic computations. In James Chapman and Paul Blain Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012*, volume 76 of *EPTCS*, pages 68–89, 2012. doi: 10.4204/EPTCS.76.7. URL https://doi.org/10.4204/EPTCS.76.7.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988. doi: 10.1145/53990.54010. URL https://doi.org/10.1145/53990.54010.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, August 2021a. Version 6.1. http://www.cis.upenn.edu/~bcpierce/sf.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, August 2021b. Version 6.1. http://www.cis.upenn.edu/~bcpierce/sf.

Maciej Piróg and Jeremy Gibbons. The coinductive resumption monad. In Bart Jacobs, Alexandra Silva, and Sam Staton, editors, *Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA, June 12-15, 2014*, volume 308 of *Electronic Notes in Theoretical Computer Science*, pages 273–288. Elsevier, 2014. doi: 10.1016/j.entcs.2014.10.015. URL https://doi.org/10.1016/j.entcs.2014.10.015.

François Pottier and Jonathan Protzenko. A few lessons from the mezzo project. In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPIcs*, pages 221–237. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi: 10.4230/LIPIcs.SNAPL.2015.221. URL https://doi.org/10.4230/LIPIcs.SNAPL.2015.221.

Jacob Prinz, G. A. Kavvos, and Leonidas Lampropoulos. Deeper shallow embeddings. In *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7 to August 10, 2022, Haifa, Israel*, LIPIcs, pages 19:1–19:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.ITP.2022.19.

Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. A unifying type-theory for higher-order (amortized) cost analysis. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi: 10.1145/3434308. URL https://doi.org/10.1145/3434308.

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *Found. Trends Program. Lang.*, 5(2-3):102–281, 2019a. doi: 10.1561/2500000045. URL https://doi.org/10.1561/2500000045.

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for proof reuse in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 26:1–26:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019b. doi: 10.4230/LIPIcs.ITP.2019.26. URL https://doi.org/10.4230/LIPIcs.ITP.2019.26.

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992*, pages 288–298. ACM, 1992. doi: 10.1145/141471.141563. URL https://doi.org/10.1145/141471.141563.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2015. doi: 10.1007/978-3-319-22102-1_24. URL https://doi.org/10.1007/978-3-319-22102-1_24.

Dana S. Scott. *Outline of a mathematical theory of computation.* Oxford University Computing Laboratory, Programming Research Group Oxford, 1970.

Dana S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976. doi: 10.1137/0205037. URL https://doi.org/10.1137/0205037.

Dana S. Scott. Domains for denotational semantics. In Mogens Nielsen and Erik Meineche Schmidt, editors, *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer, 1982. doi: 10.1007/BFb0012801. URL https://doi.org/10.1007/BFb0012801.

Dana S. Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group Oxford, 1971.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010. doi: 10.1017/S0956796809990293. URL https://doi.org/10.1017/S0956796809990293.

Lucas Silver and Steve Zdancewic. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi: 10.1145/3434307. URL https://doi.org/10.1145/3434307.

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.

Antal Spector-Zabusky. *Don't Mind the Formalization Gap: The Design and Usage of* `hs-to-coq`. PhD thesis, University of Pennsylvania, Philadelphia, PA, mUSA, 2021. URL https://repository.upenn.edu/edissertations/4250/. Publicly Accessible Penn Dissertations. 4250.

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. doi: 10.1145/3167092. URL https://doi.org/10.1145/3167092.

Antal Spector-Zabusky, Joachim Breitner, Yao Li, and Stephanie Weirich. Embracing a mechanized formalization gap. *CoRR*, abs/1910.11724, 2019. URL http://arxiv.org/abs/1910.11724.

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. doi: 10.1145/1190315.1190324. URL https://doi.org/10.1145/1190315.1190324.

Josef Svenningsson. STMonadTrans: A monad transformer version of the ST monad, 2011. URL https://hackage.haskell.org/package/STMonadTrans-0.3.1.

Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for EDSL. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming - 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2012. doi: 10.1007/978-3-642-40447-4_2. URL https://doi.org/10.1007/978-3-642-40447-4_2.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013a. doi: 10.1017/S0956796813000142. URL https://doi.org/10.1017/S0956796813000142.

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 387–398. ACM, 2013b. doi: 10.1145/2491956.2491978. URL https://doi.org/10.1145/2491956.2491978.

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.*, 4(ICFP):121:1–121:30, 2020. doi: 10.1145/3409003. URL https://doi.org/10.1145/3409003.

S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 1996. doi: 10.1007/3-540-61628-4_7. URL https://doi.org/10.1007/3-540-61628-4_7.

Wouter Swierstra. A Hoare logic for the state monad. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2009. doi: 10.1007/978-3-642-03359-9_30. URL https://doi.org/10.1007/978-3-642-03359-9_30.

Wouter Swierstra and Tim Baanen. A predicate transformer semantics for effects (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP):103:1–103:26, 2019. doi: 10.1145/3341707. URL https://doi.org/10.1145/3341707.

Paolo Torrini. Modular dependent induction in coq, mendler-style. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International*

Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings, volume 9807 of *Lecture Notes in Computer Science*, pages 409–424. Springer, 2016. doi: 10.1007/978-3-319-43144-4_25. URL https://doi.org/10.1007/978-3-319-43144-4_25.

Paolo Torrini and Tom Schrijvers. Reasoning about modular datatypes with mendler induction. In Ralph Matthes and Matteo Mio, editors, *Proceedings Tenth International Workshop on Fixed Points in Computer Science, FICS 2015, Berlin, Germany, September 11-12, 2015*, volume 191 of *EPTCS*, pages 143–157, 2015. doi: 10.4204/EPTCS.191.13. URL https://doi.org/10.4204/EPTCS.191.13.

D. A. Turner. Total functional programming. *J. Univers. Comput. Sci.*, 10(7):751–768, 2004. doi: 10.3217/jucs-010-07-0751. URL https://doi.org/10.3217/jucs-010-07-0751.

Tarmo Uustalu. Monad translating inductive and coinductive types. 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 299–315. Springer, 2002. doi: 10.1007/3-540-39185-1_17. URL https://doi.org/10.1007/3-540-39185-1_17.

Niki Vazou. *Liquid Haskell: Haskell as a Theorem Prover*. PhD thesis, University of California, San Diego, USA, 2016. URL http://www.escholarship.org/uc/item/8dm057ws.

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. doi: 10.1007/978-3-642-37036-6_13. URL https://doi.org/10.1007/978-3-642-37036-6_13.

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, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014. doi: 10.1145/2628136.2628161. URL https://doi.org/10.1145/2628136.2628161.

Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: equational reasoning in Liquid Haskell (functional pearl). 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.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL):53:1–53:31, 2018b. doi: 10.1145/3158141. URL https://doi.org/10.1145/3158141.

Dimitrios Vytiniotis, Simon L. Peyton Jones, Koen Claessen, and Dan Rosén. HALO: Haskell to logic through denotational semantics. 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 431–442. ACM, 2013. doi: 10.1145/2429069.2429121. URL https://doi.org/10.1145/2429069.2429121.

Philip Wadler. A critique of Abelson and Sussman or why calculating is better than scheming. *ACM SIGPLAN Notices*, 22(3):83–94, 1987. doi: 10.1145/24697.24706. URL https://doi.org/10.1145/24697.24706.

Philip Wadler. Recursive types for free!, July 1990. URL http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt. Draft.

Philip Wadler. Comprehending monads. *Math. Struct. Comput. Sci.*, 2(4):461–493, 1992. doi: 10.1017/S0960129500001560. URL https://doi.org/10.1017/S0960129500001560.

Philip Wadler. The expression problem, November 1998. URL http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt. Email correspondence.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989. doi: 10.1145/75277.75283. URL https://doi.org/10.1145/75277.75283.

Peng Wang, Di Wang, and Adam Chlipala. TiML: a functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.*, 1(OOPSLA):79:1–79:26, 2017. doi: 10.1145/3133903. URL https://doi.org/10.1145/3133903.

Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *J. Funct. Program.*, 18(1):87–140, 2008. doi: 10.1017/S0956796807006557. URL https://doi.org/10.1017/S0956796807006557.

Jamie Willis, Nicolas Wu, and Matthew Pickering. Staged selective parser combinators. *Proc. ACM Program. Lang.*, 4(ICFP):120:1–120:30, 2020. doi: 10.1145/3409002. URL https://doi.org/10.1145/3409002.

Li-yao Xia. Free applicative functors in Coq, July 2019. URL https://blog.poisson.chat/posts/2019-07-14-free-applicative-functors.html. Blog post.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi: 10.1145/3371119. URL https://doi.org/10.1145/3371119.

Kangfeng Ye, Simon Foster, and Jim Woodcock. Formally verified animation for RoboChart using interaction trees. In *The 23rd International Conference on Formal Engineering Methods*. Springer Science and Business Media Deutschland GmbH, June 2022. URL https://eprints.whiterose.ac.uk/188566/.

Irene Yoon, Yannick Zakowski, and Steve Zdancewic. Formal reasoning about layered monadic interpreters. *Proc. ACM Program. Lang.*, 6(ICFP), 2022. doi: 10.1145/3547630. URL https://doi.org/10.1145/3547630.

Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 71–84. ACM, 2020. doi: 10.1145/3372885.3373813. URL https://doi.org/10.1145/3372885.3373813.

Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi: 10.1145/3473572. URL https://doi.org/10.1145/3473572.

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. Verifying an HTTP key-value server with interaction trees and VST. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 32:1–32:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.ITP.2021.32. URL https://doi.org/10.4230/LIPIcs.ITP.2021.32.

Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. Handling bidirectional control flow. *Proc. ACM Program. Lang.*, 4(OOPSLA):139:1–139:30, 2020. doi: 10.1145/3428207. URL https://doi.org/10.1145/3428207.