Reusable Programming Language Components

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft
op gezag van de Rector Magnificus prof.dr.ir. T.H.J.J. van der Hagen
voorzitter van het college voor promoties
in het openbaar te verdedigen op [DATUM]

Door

Casper Roland van der REST
doctorandus in de informatica
behaald aan de Universiteit Utrecht, Nederland
geboren te Rotterdam, Nederland

[February 18, 2025 at 13:46 – version 4.2]

Voor pap, mam en alle anderen op wie ik altijd kan rekenen

CONTENTS

1 INTRODUCTION

Sometimes a computer does not do what it is supposed to do. The colloquial term for a computer mishap is *bug*, named after a famous incident in 1947 involving the Harvard Mark II computer, where operators discovered a moth that had worked its way into the internals of the machine causing it to malfunction. Although the moth in question is unlikely to cause any more trouble—it is kept safely in the National Museum of American History in Washington D.C.—the term "bug" has stuck around, and is used to refer to any unintended or unexpected behavior of a computer or program. Nowadays, with the ever-increasing pervasiveness of computers in almost all aspects of our daily lives, the number of opportunities for bugs to inflict damage is growing as well. Indeed, the economical and societal consequences of software malfunctions are potentially enormous.[1]

[1] *For example, in 2022 CISQ estimated the "cost of poor software quality" in the US to be 2.41 Trillion US dollar.*

Bugs can have a wide variety of causes, but often they arise as a result of what we might classify as miscommunication between the programmer and the machine. This class of bugs is fittingly summarized by the acronym *PEBCAK*: Problem Exists Between Chair And Keyboard. The programmer assumes they have produced a piece of code that does one thing, but when the program is executed by the machine something else happens. The program could produce a wrong result, behave unexpectedly, or even crash altogether. At first glance it is tempting to ascribe such errors to a lack of skill or training on behalf of the programmer. After all, a computer does nothing apart from diligently executing the instructions we give it. This is, however, only part of the story. The tool used by the programmer to communicate with the machine, the *programming language*, plays an important role too. It shapes the way programmers think about a problem, and sets the boundaries for what constitutes a valid program text.

There are vast differences across languages when it comes to how well they shield programmers from their own mistakes, and how effective they allow them to communicate with the machine. Undeniably, some programming languages make it easier for programmers to "shoot themselves in the foot". Over time programming languages have become more effective at protecting programmers from their own mistakes. Where the first computers were programmed by punching holes into a paper card, with careful examination of the punched card being the only safeguard against bugs, modern languages provide much more support for catching mistakes during the development process. Languages may do this by abstracting away from details of the underlying machine that are otherwise very error-prone to work with, such as memory management or concurrency, or performing sophisticated compile-time checks to eliminate certain classes of mistakes altogether. Despite these advances, bugs remain prevalent and there remains much to be gained from further improving programming language design.

1.1 THE PROBLEM: A LACK OF TYPES AND TYPE SAFETY

One of the most important techniques in programming language design for preventing bugs are *type systems*. Pierce [2002] coins the following definition:

> A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

This quote describes a somewhat classical perspective on type systems, where each (sub)term is assigned a *type* that classifies the kind of value that it evaluates to. These kinds of type systems rule out errors resulting from a mismatch in the *type of data*. For example, if we write a program that feeds a piece of text to a function that is meant to take a number as input, the *type checker*[2] should catch the mistake before the program is ever executed.

[2] *A* type checker *is a computer program that implements a type system. It is a decision procedure that determines whether a program is well-formed according to the rules of the system.*

Nowadays, type systems have the potential to statically enforce much more sophisticated properties beyond tracking the kind of values computed by a term. For example, *type-and-effect systems* keep track of the side effects of evaluating a term [Nielson and Nielson, 1999], *quantitative type theory* monitors the number of times a variable is consumed [Atkey, 2018], and *session types* controll the interaction of processes in a distributed setting [Honda et al., 1998].

It is important to realize that a type system merely defines a partion of the set of all program texts into two groups based on whether a program is well-typed according to the system or not. In practice, this judgment is only of interest to us if we can relate it to what happens when programs are executed. That is, if a program is judged to be well-typed by the system, this should imply that execution of the program is indeed devoid of certain kinds of erroneous behavior. This is an important meta-property of type systems, aptly summarized by Milner [1978] in his famous quote:

> Well-typed programs cannot go wrong.

If a type system successfully rules out all programs that exhibit a particular kind of wrong behavior,[3] it is said to be *type sound*. Dually, we could say that a type system is *complete* if it designates all programs that do not exhibit wrong behavior as well-typed. However, as a consequence of *Rice's theorem*,[4] it is impossible for a type system to be both sound and complete. Since their main purpose is to prevent software errors, type systems are typically designed to be sound, giving up completeness.

[3] *The precise meaning of "wrong" here depends on the kind of errors the type system is intended to prevent.*

[4] Rices theorem *states that any non-trivial semantic property of a program is* undecidable*: no systematic procedure can exist that decides whether such a property holds or not.*

Evidently, type soundness is a key property of a type system to be a suitable tool for helping programmers write correct programs. How can we establish that a type system is sound? The most airtight method is to mathematically verify type soundness, for example using the syntactic approach of Wright and Felleisen

[1994]. However, this is rarely done outside the context of academic research since producing the required proofs is a complex and incredibly time-consuming task. As a result, trust in the judgment of type checkers for mainstream programming languages is typically based on empirical instead of mathematical evidence.

Moreover, the absence of wrong behavior in well-typed programs is limited to those mistakes that the type system was designed to rule out. Type systems of mainstream programming languages generally model only limited aspects of a program's execution, ignoring, for example, interaction with the file system or a shared piece of memory. Still, these may very well cause erroneous behavior, so by not modelling them in the type system bugs may arise despite the type checker's blessing.

This leaves us with the following two ways in which (implementations of) mainstream programming languages are not helping programmers prevent bugs as much as they could:

- languages and their type system tend not to be formally specified, let alone verified to be sound. This diminishes the extent to which we can trust a type checker to deliver a meaningful judgment, and
- type checkers tend to model only limited aspects of a program's execution, usually the type of values computed by a term. This leaves the programmer alone in catching errors that arise from any other aspect of a program's execution.

These shortcomings do not exist because the state-of-the-art in programming language theory does not offer us ways to address them. Still, if we look at the difference between what is possible at the cutting edge of programming language design and verification, and the programming languages that people actually use, we find a gap spanning, in extreme cases, several decades. To illustrate: a form of *pattern matching* was added to both Java and Python[5] in 2021, while Haskell, which is by no means the source of the technique, included it as of its original specification released back in 1992 [Hudak et al., 1992]. This begs the question: why is there such a large gap between the cutting

[5] *More details on pattern matching in Java and Python can found in the release notes for Java 16 and Python 3.10, respectively.*

edge in programming language research and daily practice of programming language design?

THE SIMPLE, YET unsatisfying, explanation is that it is a matter of accessibility. The development of new programming languages requires a tremendous amount of time, knowledge, and effort on behalf of the language designers. The already significant cost involved with developing a programming language is multiplied when we include a formal specification of the type system and proofs of type soundness. This makes it unrealistic for most, if not all, real-world language projects to benefit from the full potential of what the state-of-the-art in programming language technology has to offer: the associated cost is simply too high. The few examples of the formal specification and verification of programming languages and their implementations at scale that do exist, such as CompCert [Leroy, 2009] or CakeML [Kumar et al., 2014], are large and multi-year colaborative efforts by experts in the field of programming languages. Considering the vast amount of time and expertise required for such projects, they present an impossible standard for the vast majority of language projects.

We summarize the situation as follows:

> The development of formally specified and verified programming languages is extremely expensive, and therefore an unrealistic standard for real-world language projects. As a result, there exists a gap between the level of safety that the cutting edge in programming language design has to offer, and what is provided by mainstream programming languages.

This thesis works towards closing this gap, by developing tools and techniques for reducing the cost of developing formally specified and verified programming languages. Ultimately, the goal is to bring the highest standard of correctness and safety offered by the state-of-the-art in programming language research to the languages that programmers use to write software every day.

1.2 SOLUTION DIRECTION AND THESIS STATEMENTS

The main strategy we adopt in this thesis for reducing the costs associated with the development of formally specified and verified programming languages is *component reuse*. That is to say: the development cost of formally verified programming languages would be greatly reduced if we could benefit from previous verification efforts by grabbing off-the-shelf components for those features that are shared with other languages. This would allow us to direct more resources towards specifying and verifying the novel features of a language project. Indeed, the POPLMark challenge [Aydemir et al., 2005], which provides a benchmark for measuring how well state-of-the art technology in formal verification is up to the task of formalizing programming language metatheory, identifies component reuse as one of the key issues inhibiting a more widespread adoption of formal verification techniques in the design and development of new programming languages.

In a perfect world, we would specify and verify programming languages on a per-feature basis. The definition of each language construct should be such that it is a mathematically rigorous and executable specification of that construct's *semantics*,[6] but simultaneously concise and perspicuous enough that it can serve as documentation to readers with a background in computer science. Furthermore, we should be able to reuse these specifications in the context of a different language project without having to alter the original specification. This combination of qualities will allow for rapid development of and experimentation with new designs of formally specified and verified programming languages.

[6] *A language's* semantics *gives a mathematical description of how a program behaves when executed.*

Before we can achieve modular, formal, and readable specifications of programming language components, we identify two fronts on which we need to make progress. The first challenge is to formally specify and verify programming language components in a way that the specification remains independent of the context of the language in which they are used, while remaining concise and readable. Especially when we require a type-safety proof to be part of this specification, and when we want to modularly define a component's side effects. Second, we recognize that the functional and/or dependently-typed *meta languages*[7] used to define and verify programming languages offer little to no support for modular definitions.[8] As a result, once we set out to define reusable programming language components in these meta languages, we find that modularity adds significant syntactic and interpretive overhead to our definitions.

[7] *Examples of this would be languages like Haskell or Agda.*

[8] Wadler [*1998*] *coined the term* expression problem *for this lack of modularity in (functional) languages.*

To MAKE THINGS concrete, this thesis explores the following two hypotheses:

> *Hypothesis 1*: reusable programming language components should be concise, readable, and safe-by-construction. Intrinsically-typed definitional interpreters are an excellent match for these requirements.
>
> *Hypothesis 2*: modularity adds significant syntactic and interpretative overhead when using state-of-the art (dependently-typed) programming languages to define reusable programming language components. Incorporating modular inductive data types in the design of functional languages is therefore a essential first step in the development of meta languages for the purpose of defining reusable programming language components.

It is important to be mindful of the limitations of intrinsically-typed interpreters as a means to specify a language's semantics. Recent work by Bach Poulsen et al. [2018], Rouvoet et al. [2020] showed that the technique is suitable for defining the semantics of imperative languages and languages with session types, but the technique has yet to be applied to more advanced type systems. While mechanizations of both System F [Chapman et al., 2019] and dependent type theory [Altenkirch and Kaposi, 2016] exist, it is not clear how their semantics could be expressed as intrinsically-typed interpreters in the style of Augustsson and Carlsson [1999]. This thesis is concerned with defining reusable programming language components for those languages to which can already be described using intrinsically-typed definitional interpreters. Before discussing in more detail how this thesis works towards this goal, we must first say a few words about how research on programming languages, including the research presented in this thesis, tends to be conducted.

1.3 RESEARCH METHODS IN PROGRAMMING LANGUAGES

Typically, papers introduce a new idea or concept[9] somewhat informally, using (code) examples and appealing to the reader's intuition. It is usually only in the later sections that papers develop a more mathematically rigorous description of their contributions, using mathematical notation and proofs. This "top-down" approach, which focusses on fostering a high-level understending of a paper's contents, is somewhat necessary in many cases due to a general lack of quantifiable measures for evaluating research. For instance, how would one quantify the usefulness of a new programming feature or proof technique? Rather than relying on the value of contributions to be self-evident, some responsibility is shifted to researches to explain why their contributions are relevant to the community.

[9] Here, "idea" can by any sort of contribution, be it a novel language design, a type-checking algorithm, techniques for specifying a language's semantics, or something else.

When it comes to mathematical proofs, the programming languages community distinguishes *pen-and-paper proofs* from *mechanized proofs*. The former documents a mathematical argument using a combination of natural language and mathematical notation, and is intended to be read by humans. The latter embeds the entire argument in a chosen formalism, and is intended to be consumed by a computer that verifies correctness of the argument.

Conferences may offer authors to submit an *artifact* in conjunction with their paper. The purpose of an artifact is to provide additional evidence for the claimed contributions of a submission, commonly in the form of a prototype implementation or a machine-checked proof. A paper and its artifact can, to some extent, be mixed using *literate programming* [Knuth, 1992]. When using literate programming, a paper intersperses pieces of human language with computer language, and the sources of the paper can be consumed by a compiler or theorem prover to validate that the programs and/or proofs are well-formed.

To evaluate the applicability of an idea design, papers may resort to small *case studies*, where ideas are applied to more involved examples compared to the minimal examples used to illustrate ideas. Such evaluations are qualitative, and often appeal to informal qualities of a solution, arguing that it is more usable, modular, or elegant. Quantitative measures are sometimes used for evaluation in papers too, if the merit of a piece of work can be judged using more easily quantifiable attributes, such as performance or code size. It is good practice for the sources of prototypes and mechanized proofs to be publicly available, such that other members can inspect and learn from them.

The individual chapters of thesis have been published at various programming language conferences, and thus follow the general practice and methodology of research in the field of programming languages as described above.

1.4 APPROACH AND THESIS STRUCTURE

The key technique for defining programming languages we embrace in this thesis are *definitional interpreters* [Reynolds, 1998]. The gist of the approach is to specify a language's semantics by defining an interpreter, and equating the dynamic semantics of the language to the behavior of this interpreter. In the context of definitional interpreters, it is important to distinguish the *object language*, the language we are defining a semantics for, and the *meta language* (sometimes also referred to as the *host language*), which is the language in which we define the semantics.

Broadly speaking, there are three main approaches when it comes to formally specifying a programming language's behavior:

1. *axiomatic semantics* [Hoare, 1969], in which we use logical predicates make assertions about a program's effect on the program state,

2. *operational semantics* [Plotkin, 2004], in which we use an inductively-defined transition relation to describe how a program evolves over time during execution, and

3. *denotational semantics* [Scott and Strachey, 1971], in which we map programs onto a mathematical object that models their semantics.

Definitional interpreters are most closely related to denotational semantics, where the "mathematical" object that we use to model programs of the object language is a program in the meta language. Of course, the mathematical rigor of such a specification depends on the meta languages used. A definitional interpreter written in Haskell, for example, cannot be taken as a mathematical specification of a language due to the presence of nontermination, exceptions, and other unsafe language features such as `unsafePerformIO`. When using a language like Agda, on the other hand, the function that maps object language terms to Agda programs must be *total*,[10] hence we can view definitional interpreters written in Agda as a denotational semantics that denotes programs of the object language as terms in Agda's type theory. We highlight that for the purpose of defining reusable programming language components, there is a balance to be struck between these extremes. While mathematical rigor is crucial, so is readability and usability of the specifications.

[10] *Here,* total *means that for every possible input value, an output will be computed in finite time.*

When working in a dependently typed language, such as Agda, *intrinsically-typed* definitional interpreters [Augustsson and Carlsson, 1999] offer a combination of readability and safety. The crux of the technique is to encode a typing invariant in the abstract syntax tree of the object language, such that it is only inhabited by well-typed terms.[11] The corresponding interpreter then maps well-typed terms to values of the same type. While the resulting definition looks like a "normal" interpreter, its type is precise enough such that we can omit any code for dealing with ill-formed terms. Due to totality of the function that implements the interpreter, the fact that it exists immediately implies a type-safety theorem about the typing invariant encoded in the abstract syntax. For these reasons intrinsically-typed interpreters

[11] *The converse, more traditional approach, is to use* extrinsic typing, *where well-typedness is defined separately as a predicate over an untyped abstract syntax type.*

are an attractive candidate for specifying and verifying reusable language components.

THIS THESIS IS organized into two parts that respectively support the two hypotheses described in Section 1.2. In part I (Chapters 2 and 3), we work on furthering the collection of techniques available to languages designers for writing type-safe modular language specifications in Agda. While this is possible, doing things modularly does add some overhead to our definitions. In part II (Chapters 4 and 5), we work towards a meta language design that has modularity built in, reducing this overhead while retaining type safety.

In Chapter 2, we discuss how to define modular intrinsically-typed interpreters in Agda. We give a definition of *intrinsically-typed language fragments* that specify the syntax and semantics of a language component in a way that is type-safe by construction, such that language fragments can be composed and reused freely to build larger languages. While modular type safety proofs have been studied before by Delaware et al. [2013a], Keuchel and Schrijvers [2013], and Delaware et al. [2013c], their work focusses on an extrinsic style of proofs, inducing considerable syntactic overhead compared to the intrinsic approach.[12]

[12] *The amount of code needed to specify a verified language component differs roughly by one order of magnitude.*

In Chapter 3, we discuss how to define modular elaborations of *higher-order effects*, along with modular reasoning principles that allow us to verify that elaborations respect equational theories of the effect(s) they implement. While there is a considerable amount of existing work on (modular) higher-order effects (for example, by Yang et al. [2022], Van den Berg et al. [2021a], and Wu et al. [2014]), these focus on defining handlers for higher-order effects directly. We take a slightly different approach by defining the semantics of higher-order effects in terms of algebraic effects, effectively following the approach by Plotkin and Pretnar [2009b] of implementing higher-order effects as handlers, but adding an additional syntactic layer to regain modularity.

In Chapter 4, we present the design of a meta language for developing reusable programming language components, that has built-in support for modular data types and effects. A crucial feature of the language is that we can solve the expression problem [Wadler, 1998] without resorting to embedding a data type's initial algebra semantics [Goguen, 1976], reducing the syntactic overhead of modularity. Furthermore, it features a type-and-effect system based on *latent effects* [Van den Berg et al., 2021a] that support, among other things, λ-abstraction as a user-defined effect.

In Chapter 5 puts the modular data types presented in Chapter 4 on formal footing by presenting a core calculus together with a type system and semantics that features built-in support for initial algebra semantics [Goguen, 1976]. Many familiar programming abstractions for modularity can be encoded in the calculus, such as modular data types and interpreters in the style of *Data Types á la Carte* [Swierstra, 2008], algebraic effects and handlers [Plotkin and Power, 2002, Plotkin and Pretnar, 2009a], as well as various implementations of higher-order effects such as *Scoped Effects* [Yang et al., 2022] or *Hefty Algebras* [Bach Poulsen and Van der Rest, 2023]. While the calculus lacks any syntactical conveniences for working with these constructions, it provides a basis for understanding how modular data types can be added to a language in a principled way.

Finally, in Chapter 6 we will look back on the work presented in these two parts, and reflect on the contributions of the thesis in light of the goals outlined in this introduction. Moreover, we will discuss future work, and how to connect the contributions presented in the two parts of this thesis.

1.5 ORIGIN OF THE CHAPTERS

The contents of this thesis correspond to previously published work as follows.

CHAPTER 2 is based on:

> Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser and Peter D. Mosses. "Intrinsically-Typed Definitional Interpreters à la Carte. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA, 2022. DOI: `10.1145/3563355`.

CHAPTER 3 is based on:[13]

Casper Bach Poulsen and Cas van der Rest. "Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects". In: *Proceedings of the ACM on Programming Languages* 7.POPL, 2023. DOI: `10.1145/3571255`.

[13] *Upon invitation, a journal version of the paper, that extends the original paper with more principled reasoning for elaborations of higher-order effects, has been submitted to the Journal of Functional Programming. At the time of writing, this version of the paper is still under review.*

CHAPTER 4 is based on:

Cas van der Rest and Casper Bach Poulsen. "Towards a Language for Defining Reusable Programming Language Components - (Project Paper). In: *Selected Papers from the 23rd International Symposium in Trends in Functional Programming*, 2022. DOI: `10.1007/978-3-031-21314-4_2`.

CHAPTER 5 IS based on:[14]

Cas van der Rest and Casper Bach Poulsen. "Types and Semantics for Extensible Data Types". In: *Proceedings of the 21st Asian Symposium on Programming Languages and Systems*, 2023. DOI: `10.1007/978-981-99-8311-7_3`.

[14] *An extended version of the paper was published on arXiv [Van der Rest and Bach Poulsen, 2023]*

The contents of the papers that form Chapters 2 to 5 of this thesis have been included verbatim in their corresponding chapter. At the start and end of each chapter, a (small) unnumbered section may have been added that discusses and reflects on the contents of the paper in the context of this thesis, and/or discussing its relation to the preceding and succeding chapters.

THE AUTHOR ALSO contributed to the following papers:

Casper Bach Poulsen, Cas van der Rest, and Tom Schrijvers. "Staged Effects and Handlers for Modular Languages with Abstraction". In: *Workshop on Partial Evaluation and Program Manipulation*, 2021.

Cas van der Rest and Wouter Swierstra. "A Completely Unique Account of Enumeration". In: *Proceedings of the ACM on Programming Languages* 6.ICFP, 2022. DOI: `10.1145/3547636`.

1.6 A NOTE ON ARTIFACTS

Part of the contributions in this thesis have been mechanized in Agda. Specifically, the contributions outlined in part I of this thesis are supported by Agda developments that are publicly available [Van der Rest et al., 2022a, Van der Rest and Bach Poulsen, 2024]. While the code examples in Part I of this thesis are type-checked, it is not the exact code contained in the accompanying developments. For expositional purposes, definitions are occasionally simplified, and (parts of) proofs and definitions may have been omitted where they would clutter the explanation. The code in the Agda development contains the full definitions.

For the language design presented in Chapter 4, there exists a prototype interpreter written in Haskell [Hudak et al., 1992] which is publicly available. The calculus design presented in Chapter 5 has not been formalized or implemented yet.

Part I

MODULAR SEMANTICS IN AGDA

2

INTRINSICALLY-TYPED DEFINITIONAL INTERPRETERS À LA CARTE

PREFACE

As noted in the introduction, intrinsically-typed definitional interpreters appear to be an excellent candidate for specifying reusable programming language components due to their conciseness and readability. Although recent work by , Bach Poulsen et al. [2018], and Rouvoet et al. [2020, 2021] has pushed the technique far beyond the typical examples of interpreters for simply-typed expression languages, they remain non-modular. There are several hurdles to overcome before we can achieve modularity of intrinsically-typed definitional interpreters, such as how to maintain type-safety in light of extensibility, how ensure that components imposing different requirements on their domain remain compatible, and how to alleviate the syntactic and interpretive overhead incurred by modularity.

This chapter begins our journey towards modular intrinsically-typed definitional interpreters by answering the first question: how do we achieve modularity of intrinsically-typed interpreters while maintaining type safety of the definitions? Since intrinsically-typed interpreters can generally be defined as a *catamorphism* over an (indexed) abstract syntax type, they are amenable to the usual techniques for encoding type-safe modularity for data types in functional languages (Section 2.2). However, in the presence of potential future extensions of the type syntax, it turns out that maintaining type safety in modular definitions requires some additional care (Section 2.3).

2.1 INTRODUCTION

Type safety is a crucial aspect of designing typed programming languages. According to Pierce [2002], "a type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." The *type safety property* of a language defines precisely what program behaviors its type system is supposed to rule out. But it is challenging to define one's type system and interpreter[15] in a way that it satisfies the intended type safety property—or, put differently, that it rules out the "bad" program behaviors it is supposed to. For this reason, programming language researchers often rely on mathematical proofs to verify that a type system and interpreter satisfies the type safety property. However, constructing a type safety proof can be a labor intensive and complex task.

[15] *It is possible to define the dynamic semantics of a language in many different ways. In this paper, we assume that the dynamic semantics is given by an interpreter.*

Our research objective in this paper is to make it as easy as possible for DSL developers to develop and verify the type safety of typed domain-specific languages (DSLs). We propose two sub-objectives that could address this goal together:

1. Support *reuse* of common programming language components, so that DSL developers can focus on developing type safe DSL components.

2. Make it easy to develop and debug type safe DSL components by automating the task of verifying that a language definition is type safe.

There is previous research that goes towards addressing these sub-objectives *individually*, but no previous research that we are aware of which provides a viable solution to *both* at the same time.

Previous work by Keuchel and Schrijvers [2013] and Delaware et al. [2013b,c] suggests a promising direction for addressing the first sub-objective, by modularizing *extrinsic* type safety proofs and interpreters. An extrinsic proof is an inductive proof on the structure of the syntax or typing rules of a language. By using the modular extrinsic proof techniques of Keuchel and

Schrijvers [2013] and Delaware et al. [2013b,c], domain-specific language designers can compose their interpreter, type system, and type safety proof from pre-proven cases for off-the-shelf components, allowing them to focus on defining and proving the type safety of domain-specific components. However, for the second sub-objective that we gave above, the extrinsic proof style has a number of shortcomings. Most importantly, it is unclear how to automate the task of constructing modular, extrinsic type safety proof cases. We also argue that the extrinsic specification style is not as easy to work with as the alternative *intrinsically-typed* style which we discuss shortly. In particular, interpreters in an extrinsic specification style contain redundant cases for bad behavior that can never happen in a type safe language because the type system rules it out. Furthermore, understanding *when and why* an extrinsic type safety proof case does not hold, is key to finding and fixing type safety errors. But it requires previous experience with inductive proofs to identify what the error is. Domain-specific language designers, however, usually do not have the necessary experience to verify type safety.

A more concise and declarative style of verifying type safety is to write an *intrinsically-typed interpreter* [Augustsson and Carlsson, 1999] in a *dependently-typed host language*. Such interpreters save language designers from having to read and write redundant cases for ill-typed expressions, as the host language type checker can automatically verify that these cases are unreachable in practice. This results in shorter, more declarative specifications that are *safe-by-construction*, in the sense that the type safety of the interpreter follows from the well-typedness of its definition. We do not have to establish type safety in a separate proof: the interpreter *is* the type safety proof. Wadler et al. [2020] observe that, in Agda [Norell, 2009], "extrinsically-typed terms require about 1.6 times as much code as intrinsically-typed", leading them to suggest that "intrinsic typing is golden".[16] Another appeal of intrinsically-typed interpreters is that language designers can debug type safety issues by using compiler errors produced by the dependently-typed host language as a guide. While the quality of compile time errors depends on the host language, it

[16] *A pun referring to the ratio in code size between intrinsically and extrinsically typed, which approximates the golden ratio.*

```
data Ty : Set where                     Val : Ty → Set
  nat  : Ty                             Val nat  = ℕ
  bool : Ty                             Val bool = Bool

                                        interp : Expr t → Val t
data Expr : Ty → Set where              interp (lit n)       = n
  lit  : ℕ → Expr nat                   interp (add e₁ e₂) = interp e₁ + interp e₂
  add : Expr nat → Expr nat → Expr nat  interp tt            = true
  tt   : Expr bool                      interp ff            = false
  ff   : Expr bool                      interp (ite e e₁ e₂)
  ite  : Expr bool → Expr t →             = if interp e then interp e₁
         Expr t → Expr t                                 else  interp e₂
```

Figure 1: An intrinsically-typed interpreter for a small expression language

does not require previous experience with inductive proof techniques. This makes intrinsically-typed interpreters an attractive approach to developing and debugging type safe languages, because it reduces the amount of work on behalf of DSL developers by taking extrinsic proof obligations for type safety and making them intrinsic to interpreter well-typing. However, intrinsically-typed interpreters fail to address our first sub-objective, since they do not in general support reuse.

In this paper we adapt and combine techniques for modular meta-theory with the intrinsically-typed approach, and develop a new notion of *intrinsically-typed language fragments* and *language fragment composition* that makes it possible to reuse off-the-shelf pre-verified components. Intrinsically-typed language fragments are as concise and declarative and similarly easy to develop and debug as plain, monolithic intrinsically-typed interpreters. Unlike monolithic interpreters, language fragments can be developed and checked in isolation and combined with other fragments to compose type safe languages from reusable components.

2.1.1 *Background: Intrinsically-Typed Interpreters*

Figure 1 shows an intrinsically-typed interpreter for a simple language with arithmetic (lit, add) and Boolean (tt, ff, ite) expressions, implemented in Agda [Norell, 2009]. It consists of:

1. a data type of object language types, Ty : Set;
2. a function that maps each object language type to a type, Val : Ty → Set;
3. an indexed data type representing the well-typed object language expressions, Expr : Ty → Set;
4. an *index-preserving evaluation function* that embeds a type safety theorem by mapping well-typed expression to a value of the same type, interp : Expr t → Val t (i.e., "well-typed expressions cannot go wrong", Milner [1978]).

The key ingredient that allows Agda to verify that the interpreter in Figure 1 is type safe is *dependent pattern matching* [Cockx, 2017, Coquand, 1992], enabling Agda to infer a precise type for each variable bound by a pattern match clause. For instance, in the clause interp (add e_1 e_2) = . . . , Agda infers that e_1 and e_2 have type Expr nat in the right hand side of the definition. Indeed, if e_1 or e_2 had any other type, the pattern match would be ill-formed according to the definition of Expr. Since e_1 and e_2 : Expr nat, Agda can deduce that the recursive calls in interp e_1 + interp e_2 must yield natural numbers, as the return type of the calls, Val nat, normalizes to ℕ. Thus no error handling of type mismatching is needed: thanks to Agda's type checker we know that *this will not happen*, and we do not have to spell out any redundant cases for going wrong.

2.1.2 *Challenge: Intrinsically-Typed Programming Language Fragments*

The interpreter in Figure 1 mixes arithmetic and Boolean expressions. If we want to extend or reuse (parts of) this language, we

```
data Ty : Set where                              data Ty : Set where
  {- ...1 -}                                       {- ...5 -}
  nat : Ty                                         bool : Ty


Val : Ty → Set                                   Val : Ty → Set
{- ...2 -}                                       {- ...6 -}
Val nat = ℕ                                      Val bool = Bool

data Expr : Ty → Set where                       data Expr : Ty → Set where
  {- ...3 -}                                       {- ...7 -}
  lit  : ℕ → Expr nat                              tt ff : Expr bool
  add : Expr nat → Expr nat → Expr nat             ite   : Expr bool → Expr t → Expr t → Expr t

interp : Expr t → Val t                          interp : Expr t → Val t
{- ...4 -}                                       {- ...8 -}
interp (lit n)        = n                        interp tt = true
interp (add e₁ e₂) = interp e₁ + interp e₂       interp ff = false
                                                 interp (ite e e₁ e₂) = if interp e then interp e₁
                                                                              else  interp e₂
```

Figure 2: Two intrinsically-typed interpreters for arithmetic expressions (left) and Boolean expressions (right).

have no option but to modify or copy-paste existing code. A better approach is to assemble interpreters from reusable components. Figure 2 provides an informal illustration of how we might define and check such reusable components in isolation, and compose them with other fragments, to incrementally develop a verified and type safe interpreter. We can compose such fragments by *concatenating* their constructor declarations and function clauses. The Agda comments {- ...$_i$ -} indicate program points where new constructors or clauses will be inserted during composition. For example, we recover the language from Figure 1 by inserting the constructors of Ty (right) at {- ...$_1$ -}, the clauses of Val (right) at {- ...$_2$ -}, and so forth. Throughout this paper we develop a semantics of intrinsically-typed language fragments that supports this kind of composition, without having to re-type-check existing language fragments.

The challenge with defining fragment composition is that not all extensions are well behaved. In particular, extensions that change the *canonical forms* of a type (i.e., removing or adding a new value constructor for an existing type) are problematic. For example, if we add Val bool = Maybe Bool at the {- ...$_6$ -} position in Figure 2, then the interp (ite e e_1 e_2) case on the right becomes ill-typed. If we were to re-type-check the composed definition, Agda would correctly reject this extended interpreter for being ill-typed. Such re-checking is, however, contrary to the goal of reusing pre-verified components. This raises the question: under which conditions is language fragment composition guaranteed to be well-typed?

This paper answers this question by introducing a subtyping relation for witnessing that canonical forms are preserved when values are extended. That is, the set of constructors for a value of a given type never changes. By using this subtyping relation in our definition of language fragments and language fragment composition, we automatically rule out bad extensions, such as the Val bool = Maybe Bool extension discussed above.

2.1.3 *Contributions*

Working in Agda[17], we make the following technical contributions:

[17] *The code in his paper is available in the accompanying artifact [Van der Rest et al., 2022a].*

- We extend (in Section 2.3) the techniques (which we recall in Section 2.2) from *data types à la carte* [Swierstra, 2008] to intrinsically-typed interpreters, and introduce a subtyping relation for canonical forms. This relation allows us to define cases of modular intrinsically-typed interpreters in a way that supports type safe composition.

- We introduce (in Section 2.4) *intrinsically-typed language fragments*, which bundle the syntax and semantics of one or more language constructs. Using a general type union, we define *canonical form unions* to support composition of language fragments with overlapping values.

- We generalize (in Section 2.5) our framework from Agda's Set to a broader class of semantic domains. With this generalization, we can modularly define languages with effects such as name binding, exceptions, and mutable state, provided that we choose a semantic domain that supports these effects upfront.
- We demonstrate (in Section 2.5) how language fragments support reuse by developing a small library of pre-verified language components, and reusing these to compose different languages.

These contributions demonstrate that modern dependently typed languages such as Agda or Idris can take us far toward addressing the two sub-objectives from the introductory paragraphs of this paper. However, it would be an overstatement to say that our contributions address the research objective of making it "as easy as possible for DSL developers to develop and verify the type safety of typed domain-specific languages (DSLs)". Our long-term ambition is to take the model that underpins our generic Agda framework, and implement it in a new meta-language that lowers the barrier for entry, and allows language designers to build verified DSLs from reusable components. The language fragment composition operation that we introduce in Section 2.4 provides a promising model for how language component reuse could work in such a meta-language.

2.2 DATA TYPES À LA CARTE

We recall how *data types à la carte* [Swierstra, 2008] lets us define *open* data types and functions. The remaining sections of this paper extend and build upon this framework. Our exposition closely follows the original exposition by Swierstra, with one difference: we encode types à la carte in Agda using *containers* [Abbott et al., 2005a, Altenkirch et al., 2015]. [18]

[18] *The reason we use containers instead of the signature functors that Swierstra [2008] uses is that the fixpoint of signature functors is not strictly positive and hence rejected by Agda (and other dependently-typed languages). The idea of using containers to implement data types à la carte in a dependently-typed language is due to Keuchel and Schrijvers [2013].*

2.2.1 *Composing Data Types*

The idea behind data types à la carte is to encode data type definitions as data. By treating data type definitions as data we can explicitly manipulate them, but also recover their meaning by mapping to Agda data types. We explain how to encode data type definitions as *signatures* that can be mapped to plain Agda data types (Section 2.2.1.1), how to compose them using *signature composition* (Section 2.2.1.2), and how to define *open data type constructors* using *signature subtyping* (Section 2.2.1.3).

2.2.1.1 *Signatures*

A signature describes a set of data type constructors. The following *record type* in Agda defines a type of signatures. We dub the record type Signature, but it corresponds to what is commonly known as a *finitary container* [Abbott et al., 2005a, Altenkirch et al., 2015].[19] We use finite containers because they make the presentation of data types more uniform (sub-expressions are always given by a vector as we illustrate next). We could have used

```
record Signature : Set where
  constructor _▷_
  field Symbols : Set
        Arity   : Symbols → ℕ
```

Signature records can be constructed using _▷_[20], as indicated by the keyword "constructor". Such records comprise a set[21] of constructor Symbols, and a function that associates an Arity (given by a natural number) with each constructor symbol.

To illustrate how signatures encode inductive data types, we compare the plain inductive definitions (top) with their encoding as signatures (bottom). Starting with arithmetic expressions: [22]

```
data ArithExpr : Set where
  lit  : ℕ → ArithExpr
  add : ArithExpr → ArithExpr → ArithExpr
```

[19] *Most sections of this paper could also be defined in terms of plain containers whose Arity is not restricted to be finite.*

[20] *The underscores in the name _▷_ indicate the argument positions of the mixfix operator.*

[21] *Set is the type of types in Agda. To rule out inconsistencies, Agda has an infinite hierarchy of Sets (i.e., Set : Set$_1$: . . .), and the Signature type that we define really lives in Set$_1$, since one of its fields is itself a Set. For presentation purposes, we abstract from universe levels and write Set everywhere.*

[22] *The notation tt ff : BoolExpr is syntactic sugar for two separate constructor declarations tt : BoolExpr and ff : BoolExpr.*

```
data ArithExprSymbols : Set where
  lit  : ℕ → ArithExprSymbols
  add : ArithExprSymbols

ArithExprΣ = ArithExprSymbols ▷ (λ where
  (lit n) → 0; add → 2)
```

And Boolean expressions:

```
data BoolExpr : Set where
  tt ff : BoolExpr
  ite   : (e e₁ e₂ : BoolExpr) → BoolExpr


data BoolExprSymbols : Set where
  tt ff : BoolExprSymbols
  ite   : BoolExprSymbols

BoolExprΣ = BoolExprSymbols ▷ (λ where
  ite → 3; tt → 0; ff → 0)
```

We recover an inductive data type from a signature by taking a fixpoint of the corresponding *signature functor*. The function ⟦_⟧ defines this functor, and the type μ its fixpoint—that is, the type of syntax trees whose constructors are given by σ. [23]

[23] *The notation* Σ[x : A] (B x) *denotes a* dependent pair *of a value* x : A *and a value of type* B x *for some* B : A → Set.

```
⟦_⟧ : Signature → (Set → Set)
⟦ σ ⟧ A = Σ[ s : Symbols σ ] (Vec A (Arity σ s))

data μ (σ : Signature) : Set where
  ⟨_⟩ : ⟦ σ ⟧ (μ σ) → μ σ
```

Taking the least fixpoint of the ArithExprΣ signature yields a type that is equivalent to ArithExpr:

```
example₀ : ArithExpr
example₀ = lit 42

example₀′ : μ ArithExprΣ
example₀′ = ⟨ lit 42 , [] ⟩

example₁ : ArithExpr
example₁ = add (lit 11) (lit 31)
```

```
example₁′ : μ ArithExprΣ
example₁′ = ⟨ add , ⟨ lit 11 , [] ⟩ :: ⟨ lit 31 , [] ⟩ :: [] ⟩
```

2.2.1.2 *Signature Composition*

Signatures can be composed by taking the disjoint union of their symbols and arities. The function :+: defines this disjoint composition using the usual sum type (⊎):

```
_:+:_ : Signature → Signature → Signature
σ₁ :+: σ₂ = (Symbols σ₁ ⊎ Symbols σ₂) ▷ (λ where
  (inj₁ s) → Arity σ₁ s
  (inj₂ s) → Arity σ₂ s)

data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

Using disjoint signature composition we can define signatures in isolation and compose them without having to re-check them. For example, we can compose the ArithExprΣ signature with BoolExprΣ, accommodating expressions that mix arithmetic and Boolean expressions. For example, the term below encodes a simple if-then-else expression (ite tt 42 0):

```
example₂ : μ (ArithExprΣ :+: BoolExprΣ)
example₂ =
  ⟨ inj₂ ite ,
      ⟨ inj₂ tt , [] ⟩
    :: ⟨ inj₁ (lit 42) , [] ⟩
    :: ⟨ inj₁ (lit 0) , [] ⟩
    :: [] ⟩
```

The repeated applications of inj_i make working with composed signatures cumbersome. Following Swierstra [2008], we address this using signature subtyping and smart constructors.

2.2.1.3 *Signature Subtyping and Smart Constructors*

Smart constructors construct instances of a data type whose full set of constructors is left open. For example, the following function constructs a literal in the syntax tree of any signature σ that contains the symbol/arity pairs of ArithExprΣ:[24]

```
lit′ : { ArithExprΣ ≼ σ } → ℕ → μ σ
```

The σ in the type of lit′ is thus decided by the context that it is used in, enabling us to flexibly reuse lit′ in different contexts. The source of this flexibility is signature subtyping:[25]

```
record _≼_ (σ₁ σ₂ : Signature) : Set where
  field inj      :  ⟦ σ₁ ⟧ A →             ⟦ σ₂ ⟧ A
        proj     :  ⟦ σ₂ ⟧ A → Maybe ( ⟦ σ₁ ⟧ A)
        proj-inj :  {x : ⟦ σ₁ ⟧ A}
                 → proj (inj x) ≡ just x
        inj-proj :  {x : ⟦ σ₁ ⟧ A} {y : ⟦ σ₂ ⟧ A}
                 → proj y ≡ just x → inj x ≡ y
```

The type $\sigma_1 \preceq \sigma_2$ witnesses that it is always possible to *inject* elements in the interpretation of σ_1 into the interpretation of σ_2, whereas the converse *projection* is only partial. The proj-inj and inj-proj fields establishes that injection and projection are partial inverses.

It is possible to automatically search for injections into co-products using instance parameters [Devriese and Piessens, 2011]. We elide the definition of the necessary instances, but they are entirely analogous to the instances found in the original data types à la carte framework [Swierstra, 2008]. The code accompanying this paper also contains the implementation.

Using signature subtyping we can implement the smart constructor for lit′ mentioned above. We use the smart inject function on below to implement a smart constructor for literals:

```
inject : { σ₁ ≼ σ₂ } → ⟦ σ₁ ⟧ (μ σ₂) → μ σ₂
inject x = ⟨ inj x ⟩

lit′ : { ArithExprΣ ≼ σ } → ℕ → μ σ
lit′ n = inject (lit n , [])
```

By defining similar smart constructors for Boolean expressions, example_2 from above can be implemented more concisely as follows:

[24] *The double braces* { . . . } *declare an instance parameter. Instance parameters are similar to type class constraints in Haskell or implicit parameters in Scala: when invoking a function with an instance parameter, an automatic search is performed during type checking time at the call-site of the function. The search will either automatically find an instance that has the correct type, or cause type checking to fail if Agda cannot find an instance parameter of the expected type.*

[25] *The curly braces in* proj-inj : {x : F A} → . . . *are implicit function arguments that Agda will attempt to automatically infer when we construct a record element. The* ≡ *type is propositional equality.*

```
example₂′ : μ (ArithExprΣ :+: BoolExprΣ)
example₂′ = ite′ tt′ (lit′ 42) (lit′ 0)
```

2.2.2 *Composing Functions*

We recall how to define a function by cases using data types à la carte.

2.2.2.1 *Algebras*

The function fold transforms a tree of type μ σ into a value of type A:

```
fold : (⟦ σ ⟧ A → A) → μ σ → A
```

The parameter of type (⟦ σ ⟧ $A \to A$) is called an *algebra*, and determines how to turn a constructor whose sub-trees are already folded into a value of type A, into a value of type A. We abbreviate the type of algebras using the following alias:

```
Algebra : Signature → Set → Set
Algebra σ A = ⟦ σ ⟧ A → A

ArithAlg : Algebra ArithExprΣ ℕ
ArithAlg (lit n ,            []) = n
ArithAlg (add  , n₁ :: n₂ :: []) = n₁ + n₂
```

The ArithAlg function defines an algebra that evaluates arithmetic expressions to natural numbers. The patterns n and m bound by matching on the add constructor are not expressions, but rather the numbers that result from evaluating the expressions in those positions. The function fold takes care of evaluating sub-expressions, and is defined as follows:

```
mutual
  fold : Algebra σ A → μ σ → A
  fold f ⟨ s , v ⟩ = f (s , map-fold f v)

  map-fold : Algebra σ A → Vec (μ σ) n → Vec A n
```

```
map-fold f []       = []
map-fold f (x :: v) = fold f x :: map-fold f v
```

To pass Agda's termination checker, we must inline the definition of map for lists (map-fold), which applies fold to recursive subexpressions.

2.2.2.2 *Algebra Composition*

We can sum algebras using the ⊕ operator given below.

```
_⊕_ : Algebra σ₁ A → Algebra σ₂ A → Algebra (σ₁ :+: σ₂) A
(f ⊕ g) (inj₁ s , v) = f (s , v)
(f ⊕ g) (inj₂ s , v) = g (s , v)
```

Summing two algebras over two signatures σ_1 and σ_2 thus yields a larger algebra for the signature composition σ_1 :+: σ_2: This algebra sum operator only works for algebras with the *same carrier type* A : Set. This implies that the ArithAlg algebra can *only* be composed with algebras that also use $\mathbb{N}$ as their carrier. This excludes, for example, the composition of ArithAlg with an algebra for Boolean expressions with Bool as its carrier type.

We can allow such compositions by defining algebras with an *open* carrier type; i.e., using signature subtyping. The idea is to represent values as signatures, and use signature subtyping to assert what value constructors each algebra *at least* requires; e.g.:

```
ArithAlg₀ :  { NatValΣ ≼ σ } → { StuckValΣ ≼ σ }
           → Algebra ArithExprΣ (μ σ)
```

The carrier of this algebra is the fixpoint of *some* signature σ, about which we only know that it contains at least the constructors described by NatValΣ and StuckValΣ. As the names suggest, NatValΣ describes natural number values, and StuckValΣ represents a *stuck* value. Stuck values are needed because the recursive positions of ArithExprΣ are not intrinsically guaranteed to return numbers, and if they do not, interpretation gets stuck. By defining a similar algebra for BoolExprΣ, we can assemble an interpreter that we can use to evaluate example₂′:

```
interpArithBool : μ (ArithExprΣ :+: BoolExprΣ)
                → μ (NatValΣ :+: BoolValΣ :+: StuckValΣ)
interpArithBool = fold (ArithAlg₀ ⊕ BoolAlg)

test : interpArithBool example₂′
       ≡ inject (nat 42 , [])
test = refl
```

2.2.3 *Discussion*

The interpreters that we can write using the techniques shown in this section are inherently weakly typed. This weak typing means that we must define values as a data type with separate constructors for each kind of a value. As a result, the algebras we define describe partial functions, returning stuck if a recursively evaluated value is not tagged with the right constructor. By contrast, the intrinsically-typed interpreter shown in the introduction is tagless: Agda uses the type index to figure out what kind of value we are dealing with, allowing us to work with bare values instead. As a result this interpreter is type-safe by construction. To define such interpreters modularly we lift data types à la carte to indexed types, defining the following types in a composable way:

```
Ty     : Set
Val    : Ty → Set
Expr   : Ty → Set
interp : ∀ {t} → Expr t → Val t
```

The key challenge is that the clauses of interp may use dependent pattern matching on values Val t at an index type t : Ty. If we know exactly what t is, we only have to consider the cases that Val associates with that type. But if Val and Ty are open-ended, how do we know that these will remain to be the only possible values for t?

The answer to this question is to ensure that values have *canonical forms*. Canonical forms lemmas are a widely-used technique for making type safety proofs robust under extension and composition [Wright and Felleisen, 1994, Pierce, 2002, Delaware et al., 2013b]. In the next section we show that this technique, in conjunction with *indexed* data types à la carte, provides exactly the abstraction we need to encode composable intrinsically-typed interpreters.

2.3 INDEXED DATA TYPES À LA CARTE, FOR DEFINING COMPOSABLE INTRINSICALLY-TYPED INTERPRETERS

We extend the data types à la carte framework to *indexed* data types, and encode *intrinsically-typed interpreters* in this framework as follows:

- We encode types Ty : Set as a plain *signature* (Section 2.3.1).
- We encode values Val : Ty → Set as an *algebra* over object language type signatures (Section 2.3.2).
- We encode expressions Expr : Ty → Set as an *indexed signature* with an *open index type* (Section 2.3.3) such that we can add new expression- or type constructors, without having to modify or re-check existing expression constructors.
- We encode interpreters interp : Expr t → Val t as an *indexed algebra* over the indexed signatures of object language expressions (Section 2.3.4). These indexed algebras have an *open carrier type* and an *open index type*, allowing us to can add new expression constructors, interpreter cases, types, and values, without modifying or re-checking existing code. It is crucial that the carrier of these indexed algebras is only open to extensions that *preserve canonical forms*.

2.3.1 *Composing Index Types*

Since the index type Ty : Set is a plain data type, we can use plain data types à la carte to encode it as a signature. For example, below on the left is a data type NatTy representing a notion of object language type with a single type constructor, and on the right is its signature encoding:

```
data NatTy : Set where
  nat : NatTy

data NatTyShape : Set where
  nat : NatTyShape

NatTyΣ = NatTyShape ▷ (λ _ → 0)
```

By similarly encoding a Boolean type constructor as a signature BoolTyΣ, we can compose a signature that encodes the object language types of the interpreter from Figure 1:

```
ArithBoolTyΣ = NatTyΣ :+: BoolTyΣ
```

2.3.2 *Composing Intrinsically-Typed Values*

The intrinsically-typed interpreter in Figure 1 defines Val : Ty → Set as a function. This function maps object language types to their *canonical forms* (i.e., the set of possible value constructors) of that type. (Note that it is also possible to model Val as a data type, but a benefit of modeling Val as a function is that values are *tag-less* [Augustsson and Carlsson, 1999], which avoids the need to tag and untag values in the interpreter.) Since Ty : Set is encoded as a signature, we can encode Val as an algebra over that signature. The following algebras define the canonical forms of nat and bool:

```
NatVal : Algebra NatTyΣ Set         BoolVal : Algebra BoolTyΣ Set
NatVal (nat , []) = ℕ               BoolVal (bool , []) = Bool
```

We can compose these algebras using the algebra sum operation from Section 2.2.2:

```
ArithBoolVal : Algebra ArithBoolTyΣ Set
ArithBoolVal = NatVal ⊕ BoolVal
```

Folding ArithBoolVal over ArithBoolTyΣ yields a function that is isomorphic to Val from Figure 1.

2.3.3 *Composing Intrinsically-Typed Expressions*

The expressions of the intrinsically-typed interpreter in Figure 1 are defined as an *indexed data type* Expr : Ty → Set whose index type is Ty. To define this type in a composable way, we lift the notion of signature from the data types à la carte framework discussed in Section 2.2 to *indexed signatures*, and obtain a framework for *indexed* data types à la carte.

2.3.3.1 *Indexed Signatures*

Below is the type of *indexed signatures* that describe *I*-indexed data types. We dub this type ISignature *I*, but this type is also commonly known as a *finitary indexed container* [Altenkirch et al., 2015]:

```
record ISignature (I : Set) : Set₁ where
  constructor _▶_
  field ISymbols : I → Set
        Indices   : {i : I} → ISymbols i → List I
```

The ISymbols field relates each index to a set of symbols, and the Indices field associates the symbols at each index with a list whose length represents the arity of each constructor symbol, and whose elements describe what the index (of type *I*) of each recursive position is. We can interpret indexed signatures as indexed data types, just as we interpreted signatures as plain data types (Section 2.2):

```
I⟦_⟧ : ISignature I → (I → Set) → (I → Set)
I⟦ ζ ⟧ P i = Σ[ s : ISymbols ζ i ] (All P (Indices ζ s))

data Iμ (ζ : ISignature I) : I → Set where
  I⟨_⟩ : {i : I} → I⟦ ζ ⟧ (Iμ ζ) i → Iμ ζ i
```

The implementation of the I⟦_⟧ function uses the following All relation on lists, which asserts that each element in a given List *I* satisfies a given proposition *P* : *I* → Set:

```
data All (P : I → Set) : List I → Set where
  []    : All P []
  _::_  : {i : I} {xs : List I} → P i → All P xs → All P (i :: xs)
```

We can think of I⟦ ζ ⟧ as mapping an indexed signature ζ to its corresponding signature functor on I-indexed types, similarly to how ⟦ σ ⟧ maps a plain signature to its corresponding signature functor on plain types.

2.3.3.2 *Indexed Signature Composition*

Two indexed signatures with the same index type I can be summed into a larger signature that comprises the shapes and indices of both:

```
_:++:_ : ISignature I → ISignature I → ISignature I
ISymbols (ζ₁ :++: ζ₂) i          = ISymbols ζ₁ i ⊎ ISymbols ζ₂ i
Indices    (ζ₁ :++: ζ₂) (inj₁ s) = Indices ζ₁ s
Indices    (ζ₁ :++: ζ₂) (inj₂ s) = Indices ζ₂ s
```

Using these ingredients we can now define composable indexed expression types corresponding to the Expr : Ty → Set from Figure 1 where the index type Ty is *fixed*. However, since new language fragments may introduce new object language type constructors, we need to model Ty in a way that allows such extensions.

2.3.3.3 *Indexed Signatures with Open Index Types*

We can define signatures whose index type is open by using the subtyping relation _≼_ (Section 2.2.1.3) to witness a lower bound on the index type, just like we did in Section 2.2.2.2. The indexed signature below thus describes intrinsically-typed arithmetic expressions whose type constructors are described by any signature σ that contains *at least* the constructors described by NatTyΣ:

```
data IArithExprSymbols {_ : NatTyΣ ≼ σ} : μ σ → Set where
  val : ℕ → IArithExprSymbols (inject (nat , []))
  add :     IArithExprSymbols (inject (nat , []))

IArithExprΣ : {_ : NatTyΣ ≼ σ} → ISignature (μ σ)
IArithExprΣ = IArithExprSymbols ▶ (λ where
  (val _) → []
  add     → inject (nat , []) :: inject (nat , []) :: [])
```

By defining Boolean expressions in a similar manner, we can compose indexed signatures with open index types:

```
IArithBoolExprΣ : { _ : NatTyΣ ≼ σ } → { _ : BoolTyΣ ≼ σ }
                → ISignature (μ σ)
IArithBoolExprΣ = IArithExprΣ :++: IBoolExprΣ
```

By defining a similar subtyping relation for indexed signatures as we did for plain signatures, we can define smart constructors for indexed types, similarly to how we defined smart constructors for plain types in Section 2.2.1.3. We elide this relation for brevity, and refer the interested reader to the code accompanying the paper where it can be found.

2.3.4 *Composing Index-Preserving Functions*

Our goal is to use indexed algebras to encode interpreters of type interp : Expr t → Val t whose index type Ty : Set and value type Val : Ty → Set are *open*. That is, we should define interp in a way that we can add new constructors to Ty : Set and Val : Ty → Set and ensure that pattern matches inside interp on values remain exhaustive. We realize this goal by defining a subtyping relation in Section 2.3.4.2 that characterizes such safe extensions. But first, we need indexed algebras.

2.3.4.1 *Indexed Algebras*

We can generically fold a tree of type Iμ ζ i into a value of type $P\ i$ using the following function, where $P : I \to$ Set:

```
Ifold : ∀[ I⟦ ζ ⟧ P ⇒ P ] → ∀[ Iμ ζ ⇒ P ]
```

Its implementation is analogous to the implementation of fold from Section 2.2.2.1, and its type uses the following abbreviations for indexed types (both from the Agda Standard Library[26]):

[26] https://github.com/agda/agda-stdlib

```
∀[_] : (I → Set) → Set
∀[_] {I} P = {i : I} → P i
```

```
_⇒_ : (I → Set) → (I → Set) → (I → Set)
(P ⇒ Q) i = P i → Q i
```

We call the function parameter ∀[I⟦ ζ ⟧ P ⇒ P] an *indexed algebra*, which we will abbreviate using the following alias:

```
IAlgebra : (ζ : ISignature I) (P : I → Set) → Set
IAlgebra ζ P = ∀[ I⟦ ζ ⟧ P ⇒ P ]
```

Just like plain algebras, indexed algebras are closed under indexed signature sums:

```
_:⊕:_ : IAlgebra ζ₁ P → IAlgebra ζ₂ P
        → IAlgebra (ζ₁ :++: ζ₂) P
(f :⊕: g) (inj₁ s , a) = f (s , a)
(f :⊕: g) (inj₂ s , a) = g (s , a)
```

This sum operation assumes that both summands have the *same* carrier, P. To sum indexed algebras corresponding to intrinsically-typed interpreters with *different* notions of types and values, we need a subtyping relation that witnesses what values (or *canonical forms*) a carrier type *at least* has.

2.3.4.2 *Canonical Forms Subtyping*

Our goal is to define indexed algebras with open carrier types in the style illustrated by interpArithAlg below:

```
interpArithAlg : { _ : NatVal ⊆ W }
               → IAlgebra IArithExprΣ (fold W)
interpArithAlg (val n ,           []) = ↑ n
interpArithAlg (add   , n₁ :: n₂ :: []) = ↑ (↓ n₁ + ↓ n₂)
```

Here, _⊆_ denotes value subtyping. In order to define composable intrinsically-typed interpreters, this subtyping relation should witness that we can *safely* convert between NatVal and W. In the definition of interpArith, we use ↑ to represent a safe “upcast” from NatVal to W, and ↓ to represent a safe “downcast” from W to NatVal. We define _⊆_ in terms of a *type isomorphism*:

```
record _↔_ (A B : Set) : Set where
  field inj↔      : A → B
```

```
proj↔     : B → A
proj-inj↔ : {a : A} → proj↔ (inj↔  a) ≡ a
inj-proj↔ : {b : B} → inj↔   (proj↔ b) ≡ b
```

The inj↔ and proj↔ fields lets us convert any A into a B and vice versa. The proj-inj↔ and inj-proj↔ fields restrict inj↔ and proj↔ to proper inverses.

Intuitively, a witness of the form $W_1 \subseteq W_2$ tells us that W_1 and W_2 define *the same values for the same types*, but W_2 may define values for more types than W_1.

```
record _⊆_ (W₁ : Algebra σ₁ Set)
           (W₂ : Algebra σ₂ Set) : Set where
  field { ≼-type } :  σ₁ ≼ σ₂
        canonical  :  {V : T → Set} → (t : ⟦ σ₁ ⟧ T)
                   → W₁ (fmap V t) ↔ W₂ (fmap V (inj t))
```

The ≼-type[27] field establishes that W_2 is defined on the same (or more) types as W_1. The canonical field asserts that W_1 and W_2 define *the same value* (up to isomorphism) for every shared type. We express this fact by requiring that W_1 and W_2 are isomorphic for all shared types t, invariant of the type of its sub-trees (T), or the way sub-trees are mapped to values (V). To apply V to the sub-trees in t, we use the function fmap:

[27] *By wrapping the ≼-type field of this record type in instance argument brackets, instance parameter search will be able to automatically resolve this field projection. For example, the inj field projection that occurs in the canonical field of the _⊆_ record type is implicitly projecting from ≼-type.*

```
fmap : (f : A → B) → ⟦ σ ⟧ A → ⟦ σ ⟧ B
fmap f (s , v) = (s , map f v)
```

Finally, we can implement the safe up-casting (↑) and down-casting (↓) operations:

```
↑ : { _ : V ⊆ W } → V (fmap (fold W) t) → fold W (inject t)
↓ : { _ : V ⊆ W } → fold W (inject t) → V (fmap (fold W) t)
```

The implementation of these function uses the type isomorphism in the canonical field of the { _ : V ⊆ W } instance argument to safely convert between the values computed by the sub- and super algebras. Comparing with Section 2.2, we can think of ↑ as the intrinsically-typed counterpart to inject, and of ↓ as the intrinsically-typed counterpart to a weakly-typed projection function. The crucial difference is that ↓ is a total function, whereas project may return nothing.

2.3.4.3 *Indexed Algebra Composition*

Indexed algebras can be composed in the same way as plain algebras. For example, we can compose interpArithAlg with the following indexed algebra for evaluating intrinsically-typed Boolean expressions:

```
interpBoolAlg :   { _ : BoolVal ⊆ W }
                → IAlgebra IBoolExprΣ (fold W)
interpBoolAlg (tt   ,                   []) = ↑ true
interpBoolAlg (ff   ,                   []) = ↑ true
interpBoolAlg (ite  , v :: v₁ :: v₂ :: []) = if ↓ v then v₁ else v₂

interpArithBoolAlg :   { _ : NatVal ⊆ W } → { _ : BoolVal ⊆ W }
                     → IAlgebra IArithBoolExprΣ (fold W)
interpArithBoolAlg = interpArithAlg :⊕: interpBoolAlg
```

Using these indexed algebras, we can now evaluate expressions given by the fixpoint of an indexed signature, using Ifold:

```
Ifold : IAlgebra ζ P → ∀[ Iμ ζ ⇒ P ]
```

This yields an interpreter that is analogous to the interpreter from Figure 1 in the introduction:

```
interpArithBool : Iμ IArithBoolExprΣ t
                → fold (NatVal ⊕ BoolVal) t
interpArithBool = Ifold interpArithBoolAlg
```

Unlike the interpreter from the introduction, this interpreter is assembled from separately-defined, reusable components—i.e., interpArithAlg and interpBoolAlg.

2.3.5 *Discussion*

This section showed how to assemble intrinsically-typed interpreters using *indexed* data types à la carte and canonical forms subtyping. While this approach allows us to assemble languages, the notion of *language fragment* that we illustrated in Figure 2 in the introduction remains informal. There are three reasons why it is useful to define language fragments as a first-class abstraction instead. The first reason is that it would allow language designers to compose languages using a single, uniform notion of *language fragment composition*, instead of the four we used in this section to compose respectively types, values, expressions, and interpreters. The second reason is that intrinsically-typed interpreters as defined in this section are not closed under composition, since indexed algebra composition "grows" the number of canonical forms subtype constraints, as interpArithBoolAlg illustrates. The last reason is that it is possible for interpreters to make *conflicting assumptions* about values, as we illustrate below.

Consider the encoding below of an intrinsically-typed interpreter which uses *ternary* Booleans, akin to the Val bool = Maybe Bool clause discussed in Section 2.1.2. The null expression below constructs the third kind of Boolean value:

```
TernaryBoolVal : Algebra BoolTyΣ Set
TernaryBoolVal (bool , []) = Maybe Bool

data INullExprShape { _ : BoolTyΣ ≼ σ } : μ σ → Set where
  null : INullExprShape (inject (bool , []))

INullExprΣ : { BoolTyΣ ≼ σ } → ISignature (μ σ)
INullExprΣ = INullExprShape ▸ λ _ → []

interpNullAlg :   { _ : TernaryBoolVal ⊆ W }
              → IAlgebra INullExprΣ (fold W)
interpNullAlg (null , []) = ↑ nothing
```

Using algebra composition, we can compose interpBoolAlg with interpNullAlg, growing the number of subtype constraints:

```
- Causes unsolved instances, even with
- overlapping-instances enabled!
{- badAlgebra  :  { _ : BoolVal ⊆ V }
               →  { _ : TernaryBoolVal ⊆ V }
               →  IAlgebra
                    (IBoolExprΣ :++: INullExprΣ)
                    (fold V)
badAlgebra  = interpBoolAlg :⊕: interpNullAlg -}
```

This is unsatisfactory: the subtype constraints of badAlgebra represent an *unsound* identification of types, because BoolVal and TernaryBoolVal define different canonical forms for the bool type. In other words, the subtype constraints *conflict*—i.e., cannot be proven—for any V : Algebra BoolTyΣ Set. The next section introduces *intrinsically-typed language fragments*, with a composition operator that avoids the issues discussed above, and enables sound intrinsically typed fragment composition.

2.4 INTRINSICALLY-TYPED LANGUAGE FRAGMENTS

In this section we introduce *language fragments*: an abstraction that bundles a set of intrinsically-typed syntax constructors with the associated cases of an interpreter. This abstraction comes with a single composition operation, *language fragment composition*, that has nice closure properties, and that subsumes the four different notions of composition that we introduced in the previous section (for object language type signatures, value typing algebras, expression data type signatures, and interpreter algebras). It thus goes towards addressing both the first and the second sub-objective stated in the introduction of this paper, by making it easier to develop type safe DSL components in a way that supports reuse.

We first discuss (Section 2.4.1) how to bundle these four components in a way that each component is defined as being *open*. We then discuss (Section 2.4.2) how to compose language fragments, and why it is necessary to allow compositions with partially-overlapping canonical forms. Finally, we present (Section 2.4.3) language fragment composition, which makes language fragments closed under composition. The idea of language fragments, however, transcends their formulation in this section, which concerns the definition and composition fragments of simply-typed expression languages. In Section 2.5, we will consider how to apply the same techniques to a more expressive semantic domain.

2.4.1 *Canons and Language Fragments*

Language fragments (Fragment) bundle a piece of intrinsically-typed syntax with its interpretation, and are parameterized over the available *canonical forms* (Canon)—i.e., a signature of object language types together with an algebra over this signature. is given by the Fragment record type below, which in turn is parameterized by a set of canonical forms as given by the Canon record type:

```
record Canon : Set where
  constructor canon
  field
    ty : Signature
    val : Algebra ty Set


record Fragment (c : Canon) : Set where
  field expr   : ISignature (μ (ty c))
        interp : IAlgebra expr (fold (val c))
```

A value of type Fragment c is a *self-contained description* of an intrinsically-typed interpreter. It is, however, defined in terms of the *fixed* set of canonical forms given by c, meaning that the composition of such fragments is limited to fragments that depend on the *same* canonical forms. To compose language fragments with *different* canonical forms, we must define them with extension of the canon in mind, similar to how we defined indexed algebras with open carriers in Section 2.3.4.2.

Because language fragments are a self-contained description of intrinsically-typed interpreters, we can view extensibility of canonical forms independent from the definition of fragments. To do this, we define the □ modifier, which *closes* a canon-indexed type over all possible future extensions:[28]

```
□ : (Canon → Set) → Canon → Set
□ P c = ∀ {c′} → { val c ⊆ val c′ } → P c′
```

We use the □ modifier to define *open* language fragments for the intrinsically-typed interpreter components from Section 2.3:

[28] *The □ notation is borrowed from the necessity modality of modal logic. Its implementation is essentially a shallow embedding of* Kripke Semantics *[Kripke, 1963] and is inspired by Allais et al. [2018].*

```
ArithFrag : □ Fragment (canon NatTyΣ
                              NatVal)
expr   ArithFrag = IArithExprΣ
interp ArithFrag = interpArithAlg

BoolFrag : □ Fragment (canon BoolTyΣ
                             BoolVal)
expr   BoolFrag = IBoolExprΣ
interp BoolFrag = interpBoolAlg
```

The □ modifier provides exactly the subtyping proof that interpBoolAlg and interpArithAlg need. Furthermore, it is always possible to transform a open fragment to a closed fragment, and extract an interpreter from a closed fragment:

```
extract : ∀[ □ P ⇒ P ]
extract ϕ = ϕ { ⊆-refl }

toInterp : (ϕ : Fragment c) → ∀[ Iμ (expr ϕ) ⇒ fold (val c) ]
toInterp ϕ = Ifold (interp ϕ)
```

2.4.2 *Fragment Composition and the Need for Partially-Overlapping Canons*

We define a composition operation for open fragments with the same canonical forms as follows:

```
fcompose-eq : ∀[ □ Fragment ⇒ □ Fragment ⇒ □ Fragment ]
expr   (fcompose-eq ϕ₁ ϕ₂) = expr   ϕ₁ :++: expr   ϕ₂
interp (fcompose-eq ϕ₁ ϕ₂) = interp ϕ₁ :⊕:   interp ϕ₂
```

In many scenarios, however, fcompose-eq alone is insufficient: it is often necessary to compose fragments with *different* canonical forms. Indeed, ArithFrag and BoolFrag have different canonical forms, so we cannot compose them into a new open fragment with fcompose-eq.

It may seem tempting to use a composition operation that *sums* the canonical forms of fragments, but this is also problematic. Say we have a fragment that defines an interpreter for a binary less-than-or-equals expression that compares the results of evaluating its arguments:

```
LeqFrag : □ Fragment (canon
                       (NatTyΣ :+: BoolTyΣ)
                       (NatVal ⊕ BoolVal))
```

A fragment that combines LeqFrag and BoolFrag by summing their canons has the following type:

```
Fragment (canon
  (NatTyΣ :+: BoolTyΣ :+: BoolTyΣ)
  (NatVal ⊕ (BoolVal ⊕ BoolVal)))
```

The resulting fragment has two *distinct* notions of Boolean types and values! As a result, an expression such as ite (leq 0 1) 42 0 would be ill-typed; the bool type of leq 0 1 is not the same bool type that ite expects. Rather, we should identify the Boolean types of the two fragments. In other words: their canonical forms are *partially overlapping*.

2.4.3 *Fragment Composition with Partially-Overlapping Canons*

We introduce a fragment composition operation for language fragments with partially-overlapping canons in three stages. First, we introduce *type unions* (Section 2.4.3.1), which precisely characterize how two types overlap. Then, we define a similar *union for canonical forms* (Section 2.4.3.2) in terms of this type union, which describes how two canons overlap. Finally, we define *fragment composition* (Section 2.4.3.3), which combines two open fragments, given a canon union that witnesses how their canonical forms are combined.

2.4.3.1 *Overlapping Unions for Types*

Figure 3 defines the Union relation, which describes how the elements of two types A : Set and B : Set map to elements in a third type C : Set, such that each element in C corresponds to an element of either A, B, or both. The correspondence is witnessed by the functions inja, injb, and from, which should be injective. [29]

There are two trivial unions which can always be constructed:

[29] *For readers wondering how general this type union is: the* Union *type is a pushout in the category of Agda* Sets*, meaning it is union-like indeed.*

```
record Union (A B C : Set) : Set where
  field
    inja  : A → C
    injb  : B → C
    from : C → These A B

    inja-inv  : ∀ {a} →
      ⟪ _≡ a , ∅ , (_≡ a) ∘ proj₁ ⟫ (from (inja a))
    injb-inv  : ∀ {b} →
      ⟪ ∅ , _≡ b , (_≡ b) ∘ proj₂ ⟫ (from (injb b))
    from-inv : ∀ {c} →
      ⟪ (_≡ c) ∘ inja , (_≡ c) ∘ injb
      , (λ (a , b) → inja a ≡ c × injb b ≡ c) ⟫ (from c)
```

```
data These (A B : Set): Set where
  this    : A →         These A B
  that    :       B → These A B
  these   : A → B → These A B

⟪_,_,_⟫ :   (A → X)
          → (B → X)
          → (A × B → X)
          → These A B → X
⟪ f , g , h ⟫ (this   a    ) = f a
⟪ f , g , h ⟫ (that      b ) = g b
⟪ f , g , h ⟫ (these a b ) = h (a , b)
```

Figure 3: The proof-relevant relation Union A B C specifies that C is the union of A and B. The types A and B can overlap in C, in which case inja a ≡ injb b for some elements $a : A$ and $b : B$. The ternary relation uses the type These and its ⟪_,_,_⟫ eliminator shown on the right.

```
union-copy     : Union A A A
union-disjoint : Union A B (A ⊎ B)
```

The function union-copy constructs a union of A with itself such that all its elements overlap with themselves. Conversely, union-disjoint constructs a union of two sets of types A and B such that none of their elements are identified in A ⊎ B. We elide the proofs of the inverse laws (inja-inv, injb-inv, and from-inv) here; the code accompanying this paper contains the details.

2.4.3.2 *Overlapping Canons*

Using the Union relation, we define a similar ternary union relation for canons:

```
record _•_≣_ (c₁ c₂ c : Canon) : Set where
  field
    { ty-union } :   Union
                       (⟦ (ty c₁) ⟧ T) (⟦ (ty c₂) ⟧ T) (⟦ (ty c) ⟧ T)
    canonicalₗ  :   {V : T → Set} {t : ⟦ (ty c₁) ⟧ T}
                  → (val c₁) (fmap V t ) ↔ (val c) (fmap V (inja t))
    canonicalᵣ  :   {V : T → Set} {t : ⟦ (ty c₂) ⟧ T}
                  → (val c₂) (fmap V t ) ↔ (val c) (fmap V (injb t))
```

The canon union is directed by a type union: ty-union witnesses that the type constructors of c are a union of the type constructors in c_1 and c_2. The key, however, are the additional proofs $\mathsf{canonical}_l$ and $\mathsf{canonical}_r$ that witness that the values of c_1 and c_2 agree in the overlap (as described by the type union) of their types. This amounts to a modularization of canonicity lemmas.

Since canon union is based on type union, we can construct similar trivial unions, where either all type constructors are overlapping (•-copy), or no type constructors are overlapping (•-disjoint, which uses the auxiliary disjoint canon union on the left):

```
_⊎ᶜ_ : (c₁ c₂ : Canon) → Canon
ty  (c₁ ⊎ᶜ c₂) = (ty  c₁) :+: (ty  c₂)
val (c₁ ⊎ᶜ c₂) = (val c₁) ⊕  (val c₂)

•-copy     : c  • c     ≡ c
•-disjoint : c₁ • c₂    ≡ (c₁ ⊎ᶜ c₂)
```

We show how to use •-disjoint and •-copy to compose ArithFrag, BoolFrag, and LeqFrag in Section 2.4.3.4.

2.4.3.3 *Fragment Composition Operation*

Using canon union, we can now define a general fcompose operation which lets us compose fragments with partially-overlapping canons:

```
fcompose :  □ Fragment c₁ → □ Fragment c₂ → c₁ • c₂ ≡ c
          → □ Fragment c
```

Perhaps surprisingly, we can implement this operation in terms of the fcompose-eq operation, by (1) exploiting the *comonadic structure* of □, and (2) observing that we can recover subtyping (_⊆_) witnesses from union (_•_≡_) witnesses. For □, we already showed that it has an extract function Section 2.4.1. Additionally, we can define a duplicate function.[30], following from transitivity of _⊆_:

```
duplicate : ∀[ □ P ⇒ □ (□ P) ]
duplicate px { w₁ } { w₂ } = px { ⊆-trans w₁ w₂ }
```

[30] *The* extract *and* duplicate *functions correspond to the operations of a* comonad.

The type of duplicate says that we can "weaken" the canon that is implicitly quantified by $\forall[\_]$,

Next, we observe that canon union implies value subtyping:

$$\bullet\text{-to-}\subseteq_l : c_1 \bullet c_2 \equiv c \to (\mathsf{val}\ c_1) \subseteq (\mathsf{val}\ c)$$

$$\bullet\text{-to-}\subseteq_r : c_1 \bullet c_2 \equiv c \to (\mathsf{val}\ c_2) \subseteq (\mathsf{val}\ c)$$

Using these ingredients, we can define fcompose in terms of fcompose-eq as follows:

```
fcompose :  □ Fragment $c_1$ → □ Fragment $c_2$ → $c_1 \bullet c_2 \equiv c$
            → □ Fragment $c$
fcompose $\phi_1$ $\phi_2$ $u$ =
  fcompose-eq
    (duplicate $\phi_1$ { $\bullet$-to-$\subseteq_l$ $u$ })
    (duplicate $\phi_2$ { $\bullet$-to-$\subseteq_r$ $u$ })
```

The duplicate function is used to "weaken" the c_1 and c_2 canons of the left and right fragment into the canon union c. The lemmas $\bullet$-to-$\subseteq_l$ and $\bullet$-to-$\subseteq_r$ witness that this weakening is safe.

2.4.3.4 *Fragment Composition Examples*

With fcompose we compose languages from fragments:

```
ArithBoolFrag = fcompose ArithFrag BoolFrag ●-disjoint
```

From this fragment, we can derive the intrinsically typed interpreter discussed in Section 3.1:

```
InterpArithBool = toInterp (extract ArithBoolFrag)
```

We can also compose ArithBoolFrag with LeqFrag, to obtain a larger language:

```
LeqArithBoolFrag = fcompose LeqFrag ArithBoolFrag ●-copy
```

In summary, fcompose addresses the three concerns we discussed in Section 2.3.5. It provides a single, uniform composition operation for language fragments with compatible but possibly different canonicity assumptions, such that language designers do not have to manually assemble types, values, expressions, and interpreters separately, using four different composition operators. It ensures that language fragments are closed under composition using fcompose. And the interpreters we extract from language fragments cannot have conflicting canonicity assumptions.

2.5 LANGUAGE FRAGMENTS WITH LEXICAL VARIABLES AND EFFECTS

Language fragments presuppose both the notion of typing and semantics. The definition of a fragment in Section 2.4 takes well-typed terms to be indexed families (Expr : Type $\to$ Set), and their semantics to be an interpreter (interp : Expr t $\to$ Val t). These presupposed notions limit the expressive power of fragments. For example, well-typed expressions with lexical binding are usually presented as indexed families that are additionally parameterized by a typing context. Their interpreter correspondingly requires an environment of values for variables. Well-typed expressions for ML-style references, on the other hand, have an intrinsically-typed interpretation in the category of monotone predicates [Bach Poulsen et al., 2018]. This motivates this section, in which we show that the notion of intrinsically-typed language fragments introduced in the previous section can be transported to more expressive semantic domains.

In this section we generalize language fragments to a class of semantic domains (Section 2.5.1) that we show can be used to define intrinsically-typed interpreters for the simply-typed λ-calculus (Section 2.5.2), exceptions (Section 2.5.3), and ML-style references (Section 2.5.4). Although intrinsically-typed language fragments can be transported to this more general setting, fragment composition only combines fragments that are interpreted into the same semantic domain. Since the examples in Sections 2.5.2 to 2.5.4 are interpreted in different domains, they cannot be combined into the same language using fragment composition. Instead, we can manually lift these fragments into a common "super domain" where they can be composed, as we demonstrate with our case study in Section 2.5.5. Furthermore, additional innovation may be required to modularize richer semantic domains that do not fit into the description we introduce in Section 2.5.1.

2.5.1 *Fragments for a Class of Semantic Domains*

We generalize the definition of language fragments to permit: (1) expressions that are typed relative to a context for de-Bruijn encoded lexical variables, (2) interpretation functions that accept a lexical environment as parameter, and (3) side effects—in particular ML-style references. The generalization is based on a generalization of the codomain of interpretation functions to some *cartesian category* $\mathcal{C}$—that is, a category with all products and a terminal object. Informally, the semantic domains presupposed by the definition of fragments from Section 2.4 vs. the definition we consider in this section differ as shown in Figure 4

	Typing	Type Semantics	Term Semantics
Section 2.4	$e : t$	Val t : Set	interp ($e : t$) : Val t
Section 2.5	$\Gamma \vdash e : t$	Val t : obj($\mathcal{C}$)	interp ($\Gamma \vdash e : t$) : $\mathcal{C}$(Env Γ, Val t)

Figure 4: Overview of how the domain of intrinsically-typed interpreters changes between Section 2.4 and this section

Here, obj($\mathcal{C}$) represents the objects of $\mathcal{C}$, and $\mathcal{C}$(Env Γ, Val t) represents the morphisms from Env Γ to Val t in $\mathcal{C}$.

In Agda, we can define a type of categories Category$_0$ and a generalized canon Canon$_0$ that maps types to objects of some category $\mathcal{C}_0$ as follows:

```
record Category₀ : Set where
  field obj     : Set
        morph : obj → obj → Set


record Canon₀ : Set where
  field ty  : Signature
        val : Algebra ty (obj 𝒞₀)
```

However, it is not possible to define a canon for ML-style reference values in this style! The problem is that the Canon$_0$ type

```
record DomainDesc : Set where
  field obj    : (T : Set) → Set
        morph : (V : T → obj T) → (X Y : obj T) → Set
```

```
record Canon : Set where                        record Fragment (c : Canon) : Set where
  constructor canon                               field
  field                                             expr   : ISignature (List (μ (ty c)) × μ (ty c))
    ty  : Signature                                 interp : IAlgebra expr
    val : ∀ {T} → ⟦ ty ⟧ (T × obj 𝒟 T)                      (λ (Γ , t) → morph 𝒟 (fold (val c))
                → obj 𝒟 T                                                 (Env c Γ)
                                                                          (fold (val c) t))
```

Figure 5: Definition of categories, canons, and language fragments. The domain description 𝒟 is a module parameter, and Env c Γ is a de Bruijn indexed environment comprising value objects whose canonical forms are given by the canon c.

assumes that we can compositionally map types onto objects of $\mathcal{C}_0$. This is not true for ML-style references. To see the issue, consider the following mock case of a refCanon : Canon_0 that we wish to define, assuming that $\mathcal{C}_0$ is a category of monotone predicates; i.e., obj $\mathcal{C}_0$ = List T → Set.

```
val refCanon (ref , V :: []) = λ Σ → ???
```

Following how references are traditionally typed [Harper, 1994, Pierce, 2002], the right hand side is supposed to witness that there exists an location of type t in the store type Σ, where t is a subterm of ref t. However, since Canon_0 defines values as plain algebras, the mock case above does not have access to the subterm t; only to the object V : obj $\mathcal{C}_0$ resulting from folding over the sub-term t.

It is possible to construct a compositional interpretation of reference types—that is, in terms of the object *V*—in the context of *semantic typing*, where types are viewed as a *set of values*. In our setting this would correspond to defining store types as a list of Sets, and mapping reference types to a proof that *V* is a member of this list. If we naively attempt to define values this way, however, their definition becomes inconsistent, since the type of these membership proofs is simultaneously bigger than and included in the set of values! Although we can escape this paradox by stratifying the interpretation of types [Ahmed et al., 2002, Ahmed, 2004], we would need to find a way to adapt the intrinsically-typed semantics for ML-style references by Bach Poulsen et al. [2018] to interpret into such a layered domain.

Instead, we opt to generalize canons and categories to reflect that the set of objects and morphisms comprise components assembled from indexed signatures and algebras. This generalized definition is shown and Figure 5. Although similar, the explicit injection of types and values means that the resulting structure (defined in the DomainDesc record) is not quite a category. We will refer to this structure as a *domain description*, using $\mathcal{D}$ to range over it. The move from categories to domain descriptions requires three generalizations:

1. Canons are generalized to use a flavor of algebra that supports *paramorphisms* [Meertens, 1992]—i.e., a recursion scheme that provides access to each sub-term both before and after we have recursively folded over it. The val field of Canon in Figure 5 shows the generalization, which maps a pair of the original sub-term of type T and its folded counterpart of type obj $\mathcal{D}$ T to an object of type obj $\mathcal{D}$ T, where T is the final set of object language types.

2. Objects of a domain description may depend on a final type of object language types, as the T : Set parameter of the obj field of DomainDesc in Figure 5 indicates. This generalization is used to define modular paramorphic algebras for which we only learn the final type of object language types after we are done composing all canons and fragments.

3. Morphisms of domain descriptions may depend on the final value typing T → obj $\mathcal{D}$ T, as the first parameter of the morph field of Fragment in Figure 5 indicates. This generalization is used to define domain descriptions whose morphisms depend on value typings. For example, to define ML-style references Section 2.5.4 we use morphisms that implicitly thread stores with values that depend on value typings.

Using these generalizations, the Fragment type in Figure 5 ensures that a fragment closure □ Fragment will use the final canon of the language as the definition of the final set of types and the final set of value typings which objects and morphisms depend on.

By deriving $\mathcal{D}$ from the category of Agda Sets (i.e, objects are types in Set and morphisms are Agda functions) we regain the framework as defined in Section 2.4, but now extended with the necessary infrastructure for variables and stores. It is straightforward to transport the fragments we developed in Section 2.4 to this more expressive setup.

Before we can define intrinsically-typed fragment instances in this more expressive setting, we also need to transport the definitions of isomorphism, canon union, and canon subtyping that we introduced in Section 2.3.1 and Section 2.4. The essential ingredient of these definitions is the notion of isomorphism. Lifting this notion isomorphism to relate objects of domain descriptions via morphisms of domain descriptions, all of the definitions from before translate straightforwardly. The safe upcasting and downcasting operations for translating between between the value typings of the fragment canon and the final fragment look more involved because of the switch to paramorphisms:

```
↑ : { _ : c₁ ⊆ c }
  → morph 𝒟 (fold (val c))
        (val c₁ (fmap (λ t → t , fold (val c) t) t))
        (fold (val c) (inject t))
↓ : { _ : c₁ ⊆ c }
  → morph 𝒟 (fold (val c))
        (fold (val c) (inject t))
        (val c₁ (fmap (λ t → t , fold (val c) t) t))
```

In the remainder of this section we illustrate how this more expressive variant of language fragments allows us to define language fragments for different language features and effects.

2.5.2 *Simply-Typed Lambda Calculus*

As our first example, we consider how to define a fragment for the simply-typed λ-calculus. We instantiate the definitions from Section 2.5.1 with a domain description based on the category of Agda Sets:

```
Sets : DomainDesc
obj   Sets T       = Set
morph Sets V A B = A → B
```

We use the following canon, interpreting types in Set:

```
data TFunShape : Set where
  fun : TFunShape

TFunΣ = TFunShape ▷ λ where
  fun → 2

funCanon : Canon
ty  funCanon = TFunΣ
val funCanon (fun , (s , V) :: (t , W) :: []) = V → W

fun′ : { TFunΣ ⪯ σ } → (s t : μ σ) → μ σ
fun′ s t = inject (fun , (s :: t :: []))
```

Here, we see the impact of using a paramorphism: the arguments to fun are not just replaced by their value (V/W), but paired with the original type (s/t) as well. The interpretation of function types is defined solely in terms of V and W, but we will need access to the uninterpreted recursive argument to define a language fragment for ML-style references (Section 2.5.4).

The signature LamExprΣ defines the three standard constructs of the λ calculus:

```
data LamExprShape { _ : TFunΣ ⪯ σ }
      : List (μ σ) × μ σ → Set where
  var : t ∈ Γ →         LamExprShape (Γ , t)
  abs :                 LamExprShape (Γ , fun′ s t)
  app : {s : μ σ} → LamExprShape (Γ , t)
```

```
LamExprΣ : { TFunΣ ≼ σ } → ISignature (List (μ σ) × μ σ)
LamExprΣ = LamExprShape ▶ λ where
  (var x)          →                              []
  (abs {Γ}{s}{t}) → (s :: Γ , t          )      :: []
  (app {Γ}{t}{s}) → (Γ      , fun′ s t ) :: (Γ , s) :: []
```

The var and abs constructors demonstrate the need for tracking a type context Γ. To reference a variable we must supply a witness $t \in \Gamma$ proving that it is in scope, and the type context of the function body is extended with the argument type s when constructing a λ-abstraction.

We then define a language fragment for the simply-typed λ-calculus as follows:

```
stlc : □ Fragment funCanon
expr   stlc = LamExprΣ
interp stlc (var x ,              []) nv = fetch x nv
interp stlc (abs   , e        :: []) nv = ↑ (λ v → e (v , nv))
interp stlc (app   , e₁ :: e₂ :: []) nv = ↓ (e₁ nv) (e₂ nv)
```

Variables are interpreted by invoking the function fetch : $t \in \Gamma \rightarrow$ Env $\Gamma \rightarrow$ fold (val c) t, which performs a safe lookup in the environment. Since function types are mapped to Agda functions, we can reuse Agda's function abstraction and application to interpret the abs and app constructors.

2.5.3 *Exceptions*

Next, we consider a language fragment for *safe division*, which raises an exception when the divisor is zero. In general, we can define effectful fragments by choosing a suitable *monad* [Moggi, 1989] that encapsulates the effects, and instantiating with a domain description based on the corresponding *Kleisli category* (which applies the monad to the target object of morphisms). For exceptions we use Maybe, which is a monad on the category Sets. The corresponding domain description is defined as follows:

```
MSets : DomainDesc
obj   MSets _     = Set
morph MSets _ A B = A → Maybe B
```

With this description, terms are interpreted as a function with type Env Γ → Maybe (fold (val c) t). This allows us to implement an interpreter for div, which returns nothing if the divisor is zero, where the function _/_ takes an (automatically inferred) proof that the divisor is greater than zero.

```
divide : □ Fragment natCanon
expr   divide = DivExprΣ
interp divide (div , m₁ :: m₂ :: []) nv = do
  v₁ ← m₁ nv ≫= ↓
  v₂ ← m₂ nv ≫= ↓
  case v₂ of λ where
    zero     → nothing
    (suc n) → ↑ (v₁ / suc n)
```

Rather than working with values of the Maybe type directly, we use Agda's do-notation[31] as syntactic sugar for monadic computation, as well as the ≫= operator which denotes monadic bind [Moggi, 1991]. Note that since the domain description is based on a Kleisli category, the result of up- and down casting (which are defined as morphisms) is now also wrapped in a Maybe.

[31] https://agda.readthedocs.io/en/v2.6.2.2/language/syntactic-sugar.html

2.5.4 *ML-Style References*

Finally, we consider how to define a language fragment for ML-style references, based on the intrinsically-typed semantics by Bach Poulsen et al. [2018]. We interpret into a domain based on a Kleisli category generated from the description ST, which has store-type-indexed Sets as objects and index-preserving functions as morphisms. We will discuss the relevant monad shortly.

```
ST : DomainDesc
obj   ST T       = List T → Set
morph ST V P Q = { Weakenable V } → ∀[ P ⇒ Q ]
```

In the definition of ST, we require access to the syntax of types (T) and their interpretation (V) to define the sets of objects and morphisms. Store types are defined in terms of T, and to interpret

ML-style references, we need to express the assumption that V is *weakenable*: every value that is well-typed relative to a given store, can also be typed with a *bigger* store.

Note that the ST description defined above admits predicates that are not monotone as objects; that is, objects are not guaranteed to be Weakenable. We could rectify this by requiring objects to be weakenable; i.e.:

```
ST′ : DomainDesc
obj      ST′ T         =
  ∃ λ (P : List T → Set) → Weakenable (const P)
morph ST′ V P Q = ∀[ proj₁ P ⇒ proj₁ Q ]
```

However, that would clutter the resulting interpreter, which relies on dependent pattern matching on objects. For that reason, we use ST which has less structure, but which allows us to explicitly assume that predicates are weakenable when we need it.

The canon for references shows why val needs to be a paramorphism: the interpretation of the type ref t is a proof of the form $t \in \Sigma$, which makes val non-compositional.

```
data TRefShape : Set where
  ref unit : TRefShape
TRefΣ = TRefShape ▷ λ where
  ref   → 1
  unit → 0

refCanon : Canon
ty   refCanon = TRefΣ
val refCanon (ref   , (t , V) :: []) Σ = t ∈ Σ
val refCanon (unit ,              []) Σ = ⊤
```

Interpreting ML-style references has side effects, in the sense that the interpreter can modify a global store. To define the interpreter for ML-style references, we require a monad over store predicates that encapsulates this interaction with the store. Rather than settling on a particular monad, we keep it abstract, and assume that it satisfies the Mem interface (Figure 6, left), which provides the operations alloc, retrieve, and write. Following Bach Poulsen et al. [2018], we also require that the chosen monad

```
Ref : T → List T → Set
Ref t Σ = t ∈ Σ


record Mem (M : Monad ST) : Set where
  field
    alloc    : ∀[ V t ⇒ M V (Ref t) ]
    retrieve : ∀[ Ref t ⇒ M V (V t) ]
    write    : ∀[ Ref t ⇒ V t ⇒ M V U ]
```

```
refs : { Mem M } → □ Fragment refCanon
expr refs = RefExprΣ
interp refs (init , m :: []) nv = do
  v ← m nv
  alloc v ≫= ↑
interp refs (deref , m :: []) nv = do
  l ← m nv ≫= ↓
  retrieve l
interp refs (update , m₁ :: m₂ :: []) nv = do
  (l , nv ) ← ( m₁ nv ≫= ↓ ) ^ nv ⟨ wk-env ⟩
  (v , l  ) ←   m₂ nv          ^ l  ⟨ wk-ref  ⟩
  write l v ≫= ↑
```

Figure 6: Definition of the Mem interface, and a fragment for ML-style references

has *tensorial strength* [Moggi, 1991], meaning it supports the following operation, where $\cap$ is the usual product type lifted to predicates:

$$\_\hat{}\_ : \forall[\, M\ P \Rightarrow Q \Rightarrow M\ (P \cap Q)\,]$$

We use this operation whenever we compute a value, but need to perform more computations before we can return that value. This situation occurs, for example, in the update case of the interpreter in Figure 6. Since computations may change the store, it is not immediately clear that previously computed values are still typeable relative to the updated store after running these computations. The strength operation allows us to pass these values back into the monad by pairing them with a computation, updating their store typing along the way. The key to implementing strength is to assume that the store only ever increases monotonically during execution (i.e., values can be added or changed, but never deleted), and to require that strength can only be applied to values that are weakenable with respect to this ordering. The interpreter itself (Figure 6, right) is then defined in terms of the operations provided by these interfaces, and defines interpretation for init, deref, and update expressions, which respectively create, read from, and update a reference. Hence, we can use this fragment with any monad that satisfies the Mem interface and has tensorial strength.

2.5.5 *Case Study*

To evaluate our approach we defined a small library of fragments as a case study.[32] 7 shows an overview of the Canons, monadic interfaces, Fragments, and languages that we implemented. Nodes are Agda modules, and dashed arrows are imports. For each module, we indicate the line count of the corresponding file. In addition to the Mem interface from Section 2.5.4, we assume three additional monadic interfaces: Lambda (which provides operations for function abstraction and application), General (which provides an operation for general recursion), and Except (which provides operations for throwing and catching exceptions). It is possible to construct many more languages than shown in the figure, since any unique combination of fragments can be composed into a unique language.

[32] *The case study is available as part of the accompanying artifact [Van der Rest et al., 2022a].*

The combination of ML-style references and functions means that, even without assuming the General interface, we can encode general recursion using Landin's Knot. Thus, our interpreter must also be able to assign a semantics to non-terminating programs. A common technique for representing (potentially) non-terminating computations in a total language like Agda, that we also use for our case study, is to use a *fueled interpreter* [Amin and Rompf, 2017, Owens et al., 2016]. A computation that may either return an A or diverge, is represented as a function of type $\mathbb{N} \to$ Maybe A that returns a just if it finishes computing before running out of fuel, and nothing otherwise.

The type of fragment composition necessitates that we define all the fragments from this case study using the same semantic domain. To support a fragment for ML-style references, this must be the same semantic domain that we developed in Section 2.5.4, hence, we cannot use the exact fragments for the simply-typed lambda calculus and exceptions that we defined in Section 2.5.2 and Section 2.5.3. Instead we must re-define them to interpret into the new semantic domain. To extract an interpreter for com-

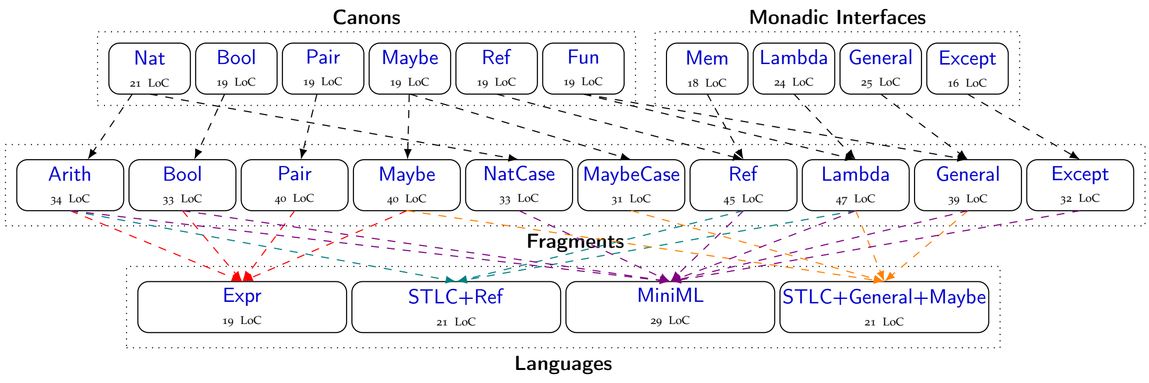

Figure 7: Overview of canons, interfaces, fragments, and languages, together with their dependencies

posed languages, we must also provide a monad that instantiates monadic interfaces, such as the Mem interface in Figure 6. Our case study achieves this by using a monad that simultaneously instantiates *all* of the interfaces in Figure 7.

The Lambda fragment maps a lambda expression to a monadic operation which accepts a monadic operation (the function body) as input, and produces a closure value as output. Choosing a monad that has *both* this operation *and* satisfies the Mem interface discussed in Section 2.5.4 requires some care. We cannot use Agda functions to represent function values, as this results in a *recursive domain equation*: a mutual dependency between values and stores which Agda's termination checker will (rightly) reject. To solve this problem, we adopt a flavor of effect handler that is similar to the *latent effect handlers* of Van den Berg et al. [2021b].

2.5.6 *Discussion*

We set out to show that our approach to intrinsically-typed and intrinsically compositional language fragments can be transported from the rather simple semantic domain for which we presented the ingredients in Section 2.4 to a much richer semantic domain from the state-of-the-art in intrinsically-typed interpreters. The key idea of our approach is that the extrinsic proof obligations of both type safety and fragment composition can be made intrinsic to fragment well-typing. We argued that the benefits of this approach are that it helps language component developers to get their semantics right from the start, and that it reduces code size. Our case study indeed demonstrated that these benefits are upheld when we employ the same approach in the relevant semantic domain.

At the same time, the case study also showed that the specifics of particular semantic domains come with their own challenges regarding compositionality. Furthermore, reuse is limited to the chosen semantic domain, which is selected upfront. To use the presented model for compositional fragments as the underpinning for a meta-language for DSL development, one needs to select a class of domains that is expressive enough to cover a large set of desired DSLs. Making this selection and tackling the compositionality challenges that are specific to that class of domains is not addressed by this paper.

2.6 RELATED WORK

We have presented an approach to constructing type-safe languages from composable, type-safe language fragments. As described in previous sections, our approach builds on *data types à la carte* [Swierstra, 2008]. Here, we describe other related work.

2.6.1 *Meta-Theory à la Carte.*

Our work is closely related to previous work on *Meta-Theory à la Carte* (MTC) [Delaware et al., 2013b], *Modular Monadic Meta-Theory* (3MT) [Delaware et al., 2013c], and *Generic Data Types à la Carte* (GDTC) [Keuchel and Schrijvers, 2013].

Delaware et al. [2013b] and Keuchel and Schrijvers [2013] focus on *pure* language features, and support binders by using *parameterized higher-order abstract syntax* [Chlipala, 2008]. Other than binding, they do not consider effectful language features. Delaware et al. [2013c] extend the MTC approach with effects, using monadic interfaces similar to Mem in Figure 6. They also construct their monads modularly using *monad transformers* [Liang et al., 1995b]. We did not build our monad modularly, as that would require composing different domain descriptions, which is an open question.

We need on average 37 LoC (counted with Al Danial's `cloc` tool) to implement verified language fragments. If we also count lines of code for definitions of canonical forms and monadic interfaces, this number approximately doubles. In comparison, MTC and GDTC report needing respectively on average 1100 LoC and 1050 LoC to define and verify similar language features. In other words, the difference in code size between our intrinsically-typed approach and the extrinsic approach found in previous work is about an order of magnitude.

Our framework code is also more concise: we use 859 LoC, whereas MTC uses 2500 LoC, GDTC uses 3500 LoC, and 3MT uses 4400 LoC. We use the Agda Standard Library for lists, relations over lists, functions for working with functions and predicates, and more. Previous works also seem to use Coq's standard library, but perhaps to a lesser extent.

Some of the difference in code sizes can be ascribed to code that deals with "wrong" cases, since these cases are absent in intrinsically-typed interpreters. Our language fragment abstraction also saves code when composing languages. This abstraction does not exist in the MTC frameworks, where language features are the sum of their parts (i.e., types, expressions, values, type system, interpreter, and type safety proof). In our framework all of these parts are summed using a single operation: fragment composition.

2.6.2 *Generic Programming and Meta-Theory*

A key technique in our work is to encode data type descriptions as signatures. The Signature type we used as our data type description is known as a *container* [Abbott et al., 2005a, Altenkirch et al., 2015]. The *universe of syntaxes* encoding [Chapman et al., 2010b, Dagand, 2013b] (which we will call just "universe encoding") encodes data type descriptions in a different way but encodes the same class of data types as containers do. The difference is in how the encoding is defined: a universe encoding is akin to the syntactic representation of a grammar, whereas the container encoding corresponds to a bag of symbols with associated arities. For generic programming applications that operate on the *syntax of data types itself*, the universe encoding is often preferred. For example, *ornamentation* is a technique for "decorating" data types with additional structure [McBride, 2011, Dagand and McBride, 2014, Ko and Gibbons, 2017, Dagand, 2017].

Allais et al. [2018] use the universe encoding to implement a generic framework for *syntaxes with variable binding*, enabling binding-aware generic programming, with generic correctness guarantees. Whereas our paper focuses on the problem of defining an intrinsically-typed interpreter in a modular way, the work of Allais et al. [2018] suggests a promising direction for meta-theoretical reasoning about object languages and interpreters defined using similar generic programming techniques as our framework. Whether it is possible to apply the generic meta-theoretical reasoning techniques of Allais et al. [2018] to the style of modular intrinsically-typed interpreters we develop in this paper is an interesting question for future work.

Final tag-less interpreters [Carette et al., 2009b] represent a different flavor of generic programming for implementing type safe definitional interpreters. In particular Carette et al. [2009b] showed that it is possible to implement type safe interpreters by modeling object language expression constructors as an abstract interface which, behind the scenes, is constructing a meta-language program that corresponds to interpreting the object language expression. The resulting interpretation of the object language expression is type safe because meta-language programs are type safe. *Object algebras* [d. S. Oliveira and Cook, 2012] are based on a similar idea, but for object oriented meta-languages. Bahr and Hvitved [2012b] extend the final tag-less approach to intrinsically-typed definitional interpreters in Haskell. Their intrinsically-typed interpreter does not treat value types as open, but does support open object language types, by using Haskell types as object language types. A similar result was achieved by

Parreaux et al. [2019] in Scala. It is unclear how the final tag-less approach can be used to define type safe semantics of languages with effects that are not built into the meta-language.

2.6.3 *Other Approaches to Modular Semantics and their Proofs*

Cimini et al. [2020] extrinsically verify type safety of language specifications by developing a meta type-system for type-safe language specifications. A (meta) type-correct specification yields a type-safe semantics. Language definitions are monolithic, and cannot be constructed from separately checked fragments. Their approach uses reduction semantics, which has some modularity issues. For example, if we add support for exception handlers, we need to copy-paste the current reduction context definition to express a context up-to the closest handler. On the other hand, their approach make it easy to verify type safety of new extensions, since that is done automatically by their meta-language type checker.

Schwaab and Siek [2013] present an Agda formalization of a modular extrinsic type safety proof for a small-step operational semantics defined using a variant [Norell, 2008] of the universe encoding due to Chapman et al. [2010b], Dagand [2013b] discussed above. The approach is closely related to the extrinsic modular proofs found in MTC [Delaware et al., 2013c], but is based on modular progress/preservation lemmas about a small-step transition relation. A direct comparison with MTC is difficult because of the difference in meta- and object language (Schwaab and Siek [2013] encode a simple object language with just numbers and lists), but we expect that progress/preservation style requires more LOC per feature. On the other hand, progress/preservation lemmas are a time tested paradigm for type safety proofs. As shown by Wadler et al. [2020], it is possible to marry the intrinsically-typed approach with the small-step style. We expect that it is possible to modularize intrinsically-typed small-step semantics using similar techniques as we use in our framework, but leave verification of this expectation for future research.

Modular Structural Operational Semantics (MSOS) [Mosses, 2004] is a framework for modularly specifying small-step operational semantics. Madlener et al. [2011] show how to implement this framework in the Coq proof assistant, and how to do modular proofs about the small-step transition relation. In a related line of work, Torrini and Schrijvers [2015] describe a different method for modular proofs in Coq, and provide, as case study, a modular extrinsic type safety proof about a modularly-specified small-step transition relation. Churchill et al. [2015] use an extension of MSOS [Churchill and Mosses, 2013] to modularly specify the static and dynamic semantics of a collection of individual fundamental programming constructs, but do not establish type safety.

Interaction Trees [Xia et al., 2020] are a general purpose coinductive data structure for representing effectful and non-terminating computations based on the *freer monad* [Kiselyov et al., 2013, Kiselyov and Ishii, 2015]. As such they support modular reasoning by defining the "visible events" (i.e., effects) of a computation as a functor signature, with the type of interaction trees being closed under the co-product of these signatures. In later work, Zakowski et al. [2021] used interaction trees to give a formal semantics for LLVM IR. The flavor of effect handlers that we used to define a semantics for MiniML in Section 2.5.5 bears some semblance to interaction trees, but to explore their relation in more detail is a subject of further study.

In his popular textbook on *Types and Programming Languages*, Pierce [2002] makes use of *canonical forms* lemmas to make type safety proofs *robust* as new language fragments are gradually introduced. He attributes the idea of using canonical forms lemmas to Bob Harper (no citation given). We have shown how to define interpreters and language fragments in a way that guarantees that the subset of values that an interpreter- or language fragment knows about is guaranteed to be canonical, essentially making language fragments *robust by construction*.

2.7 conclusion

In this paper, we presented a framework for defining composable and safe by construction language fragments that can be checked in isolation and safely reused to build type safe languages. This makes it easier to develop and reuse interpreters that do not go wrong, and reduces the overhead traditionally associated with modular verification of type safety. This makes mechanized meta-theory available to a wider audience of DSL developers.

postscript

The contributions presented in this chapter indeed give us a handle on understanding how the type safety invariant encoded in intrinsically-typed interpreters can be maintained under composition. While this is an important first step towards employing intrinsically-typed interpreters for defining reusable language components, there is still work ahead. As pointed out in Section 2.5.6, language constructs may impose different requirements on the semantic domain they denote into, and it is still an open question how to mediate these different requirements when composing fragments.

Differences in the semantic domain of fragments arise, for the most part, because their interpreters exhibit different *side effects*. A modular treatment of these side effects is thus a key ingredient for composing fragments with different domains. In the next chapter, we will work towards expanding the set of side effects that can be dealt with modularly.

3 HEFTY ALGEBRAS: MODULAR ELABORATIONS AND REASONING FOR PROGRAMS WITH HIGHER-ORDER EFFECTS

PREFACE

A key question left open at the end of last chapter was how to mediate language fragments with different semantic domains. By defining the semantics of fragments in terms of operations of some abstract monad, we reduced the problem to finding a structured and modular approach to constructing these monads. Although algebraic effects and handlers [Plotkin and Pretnar, 2009a] provide a principled method for defining the syntax and implementation of abstract monads, the approach is not expressive enough to construct the monad(s) we need to implement the language fragments presented in Section 2.5.

There are two main reasons why algebraic effects and handlers are not sufficient. The first is that they are, by definition, restricted to describing the syntax of monads that only have first-order operations. That is, they cannot describe monadic operations which take computations as an argument, precluding many common operations such as exception catching. Several of the fragments shown in Figure 7, such as Lambda and Except, are implemented in terms of such higher-order operations. The second reason is that fragments may be denoted into a monad on a category with more structure than Agda's Set. The Ref fragment in Figure 7, for example, requires a monad in a category of montone predicates to enforce well-formedness of references. This chapter contributes a new approach to defining the syntax and semantics of monads with higher-order operations by elaborating them into algebraic effects.

3.1 INTRODUCTION

Defining abstractions for programming with side effects is a research question with a long and rich history. The goal is to define an interface of (possibly) side effecting operations where the interface encapsulates and hides irrelevant operational details about the operations and their side effects. Such encapsulation makes it easy to refactor, optimize, or even change the behavior of a program, by changing the implementation of the interface.

Monads [Moggi, 1989] have long been the preferred solution to this research question. However, *algebraic effects and handlers* [Plotkin and Pretnar, 2009b] are emerging as an attractive alternative solution, due to the modularity benefits that they provide. However, these modularity benefits do not apply to many common operations that take computations as arguments.

3.1.1 *Background: Algebraic Effects and Handlers*

To understand the benefits of algebraic effects and handlers and the modularity problem with operations that take computations as parameters, we give a brief introduction to algebraic effects, based on the effect handlers tutorial by Pretnar [2015]. Readers familiar with algebraic effects and handlers are encouraged to skim the code examples in this subsection and read its final paragraph.

Consider a simple operation *out* for output which takes a string as argument and returns the unit value. Using algebraic effects and handlers its type is:

$$\mathit{out} : \mathit{String} \to () \,!\, \mathit{Output}$$

Here *Output* is the *effect* of the operation. In general $A \,!\, \Delta$ is a computation type where A is the return type and Δ is a *row* (i.e., unordered sequence) of *effects*, where an *effect* is a label associated with a set of operations. A computation of type $A \,!\, \Delta$ may *only* use operations associated with an effect in Δ. An effect can generally be associated with multiple operations (but not the other way around); however, the simple *Output* effect that we consider is only associated with the operation *out*. Thus $()\,!\,\mathit{Output}$ is the type of a computation which may call the *out* operation.

We can think of *Output* as an interface that specifies the parameter and return type of *out*. The implementation of such an interface is given by an *effect handler*. An effect handler defines how to interpret operations in the execution context they occur in. The type of an effect handler is $A\,!\,\Delta \Rightarrow B\,!\,\Delta'$, where Δ is the row of effects before applying the handler and Δ' is the row after. For example, here is the type of an effect handler for *Output*:

$$\mathit{hOut} : A\,!\,\mathit{Output}, \Delta \Rightarrow (A \times \mathit{String})\,!\,\Delta$$

The *Output* effect is being handled, so it is only present in the effect row on the left.[33] As the type suggests, this handler handles *out* operations by accumulating a string of output. Below is the handler of this type:

$$\begin{aligned}\mathit{hOut} = \textbf{handler}\,\{\ & (\textbf{return}\ x) \mapsto \textbf{return}\ (x, \text{“”})\\ & (\mathit{out}\ s; k) \mapsto \textbf{do}\ (y, s') \leftarrow k\ ();\\ & \quad \textbf{return}\ (y, s \mathbin{+\!\!+} s')\ \}\end{aligned}$$

The **return** case of the handler says that, if the computation being handled terminates normally with a value x, then we return a pair of x and the empty string. The case for *out* binds a variable s for the string argument of the operation, but also a variable k representing the *execution context* (or *continuation*). Invoking an operation suspends the program and its execution context up-to the nearest handler of the operation. The handler can choose to re-invoke the suspended execution context (possibly multiple times). The handler case for *out* above always invokes k once. Since k represents an execution context that includes the current handler, calling k gives a pair of a value y and a string s', representing the final value and output of the execution context. The result of handling *out* s is then y and the current output (s) plus the output of the rest of the program (s').

[33] *Output could occur in Δ too. This raises the question: which Output effect does a given handler actually handle? We refer to the literature for answers to this question; see, e.g., the row treatment of Morris and McKinna [2019b], the* effect lifting *of Biernacki et al. [2018], and the* effect tunneling *of Zhang and Myers [2019].*

In general, a computation $m : A ! \Delta$ can only be run in a context that provides handlers for each effect in Δ. To this end, if $\Delta = \Delta_1, \Delta_2$ and $h : A ! \Delta_1 \Rightarrow B ! \Delta_1'$, then the expression $(\textbf{with } h \textbf{ handle } m) : B ! \Delta_1', \Delta_2$ runs m in the context of the handler h. For example, consider:

$$\begin{aligned}&\mathit{hello} : () \,!\, \mathit{Output}\\&\mathit{hello} = \mathit{out}\ \text{“Hello”};\ \mathit{out}\ \text{“ world!”}\end{aligned}$$

Using this, we can run *hello* in a scope with the handler *hOut* to compute the following result:

$$(\textbf{with}\ \mathit{hOut}\ \textbf{handle}\ \mathit{hello}) \equiv ((), \text{“Hello world!”})$$

An attractive feature of algebraic effects and handlers is that programs such as *hello* are defined *independently* of how the effectful operations they use are implemented. This makes it is possible to refine, refactor, or even change the meaning of operations without having to modify the programs that use them. For example, we can refine the meaning of *out without modifying the hello program*, by using a different handler *hOut′* which prints output to the console. However, some operations are challenging to express in a way that provides these modularity benefits.

3.1.2 *The Modularity Problem with Higher-Order Operations*

Algebraic effects and handlers provide limited support for operations that accept computations as arguments (sometimes called *higher-order operations*). The limitation is subtle but follows from how handler cases are typed. Following Plotkin and Pretnar [2009b], Pretnar [2015], the left and right hand sides of handler cases are typed as follows:

$$\textbf{handler}\,\{\cdots\ (\mathit{op}\ \underbrace{v}_{A};\ \underbrace{k}_{B \to C\,!\,\Delta'}\ ) \mapsto \underbrace{c}_{C\,!\,\Delta'},\ \cdots\}$$

Here it is only k whose type is compatible with the right hand side. In theory, the parameter type v would also be compatible if $A = C ! \Delta'$. However, encoding computations as parameters in this way is non-modular. The reason is that effect handlers are not applied recursively to parameters of operations [Plotkin and Pretnar, 2009b, Pretnar, 2015]; i.e., if h handles operations other than *op*, then

$$\begin{aligned}&\textbf{with } h \textbf{ handle } (\textbf{do } x \leftarrow \mathit{op}\ v; m)\\ \equiv\ &\textbf{do } x \leftarrow \mathit{op}\ v; (\textbf{with } h \textbf{ handle } m)\end{aligned}$$

This implies that the only way to ensure that v has type $A = C\,!\,\Delta'$ whose effects match the context of the operation (e.g., $k : B \to C\,!\,\Delta'$), is to apply handlers of higher-order effect encodings (such as *op*) before applying other handlers (such as h). In turn, this means that programs can contain at most one higher-order effect encoded in this way (otherwise, which handler do we apply first?). Consequently, encoding computation parameters in terms of the value v carried by an operation does not support modular definition, composition, and handling of higher-order effects.

A consequence of this restriction is that algebraic effects and handlers only support higher-order operations whose computation parameters are *continuation-like*. In particular, for any operation $\mathit{op} : A\,!\,\Delta \to \cdots \to A\,!\,\Delta \to A\,!\,\Delta$ and any $m_1, \ldots, m_n$ and k,

$$\begin{aligned}&\textbf{do } x \leftarrow (\mathit{op}\ m_1 \ldots m_n); k\ x\\ \equiv\ &\mathit{op}\ (\textbf{do } x_1 \leftarrow m_1; k\ x_1) \ldots (\textbf{do } x_n \leftarrow m_n; k\ x_n)\end{aligned} \qquad (\dagger)$$

This property, known as the *algebraicity property* [**?**], says that the computation parameter values $m_1, \ldots, m_n$ are only ever run in a way that *directly* passes control to k. Such operations can without loss of generality or modularity be encoded as operations *without computation parameters*; e.g., $\mathit{op}\ m_1 \ldots m_n = \textbf{do } x \leftarrow \mathit{op}'\ (); \mathit{select}\ x$ where $\mathit{op}' : () \to D^n\,!\,\Delta$ and $\mathit{select} : D^n \to A\,!\,\Delta$ is a function that chooses between n different computations using a data type D^n whose constructors are $d_1, \ldots, d_n$ such that $\mathit{select}\ d_i = m_i$ for $i = 1..n$. Some higher-order operations obey the algebraicity property; many do not. Examples of operations that do not include:

- Exception handling: let *catch* $m_1\ m_2$ be an operation that handles exceptions thrown during evaluation of computation m_1 by running m_2 instead, and *throw* be an operation that throws an exception. These operations are not algebraic. For example,

$$\textbf{do}\ (\mathit{catch}\ m_1\ m_2); \mathit{throw}\ \not\equiv\ \mathit{catch}\ (\textbf{do}\ m_1; \mathit{throw})\ (\textbf{do}\ m_2; \mathit{throw})$$

- Local binding (the *reader monad* [Jones, 1995]): let *ask* be an operation that reads a local binding, and *local* r m be an operation that makes r the current binding in computation m. Observe:

  $$\textbf{do}\ (\mathit{local}\ r\ m); \mathit{ask}\ \not\equiv\ \mathit{local}\ r\ (\textbf{do}\ m; \mathit{ask})$$

- Logging with filtering (an extension of the *writer monad* by Jones [1995]): let *out* s be an operation for logging a string, and *censor* f m be an operation for post-processing the output of computation m by applying $f : \mathit{String} \to \mathit{String}$.[34] Observe:

  $$\textbf{do}\ (\mathit{censor}\ f\ m); \mathit{out}\ s\ \not\equiv\ \mathit{censor}\ f\ (\textbf{do}\ m; \mathit{out}\ s)$$

[34] *The censor operation is a variant of the function by the same name the widely used Haskell* `mtl` *library.*

It is, however, possible to elaborate higher-order operations into more primitive effects and handlers. For example, *censor* can be elaborated into an inline handler application of *hOut*:

$$\begin{aligned}
&\mathit{censor} : (\mathit{String} \to \mathit{String}) \to A \,!\, \mathit{Output}, \Delta \to A \,!\, \mathit{Output}, \Delta \\
&\mathit{censor}\ f\ m = \textbf{do}\ (x, s)\ \leftarrow (\textbf{with}\ \mathit{hOut}\ \textbf{handle}\ m); \\
&\qquad\qquad\qquad\ \ \mathit{out}\ (f\ s); \\
&\qquad\qquad\qquad\ \ \textbf{return}\ x
\end{aligned}$$

The other higher-order operations above can be defined in a similar manner.

Elaborating higher-order operations into standard algebraic effects and handlers as illustrated above is a key use case that effect handlers were designed for [Plotkin and Pretnar, 2009b]. However, elaborating operations in this way means the operations are not a part of any effect interface. So, unlike plain algebraic operations, the only way to refactor, optimize, or change the semantics of higher-order operations defined in this way is to modify or copy code. In other words, we forfeit one of the key attractive modularity features of algebraic effects and handlers.

This modularity problem with higher-order effects (i.e., effects with higher-order operations) was first observed by Wu et al. [2014] who proposed *scoped effects and handlers* [Wu et al., 2014, Piróg et al., 2018, Yang et al., 2022] as a solution. Scoped effects and handlers have similar modularity benefits as algebraic effects and handlers, but works for a wider class of effects, including many higher-order effects. However, Van den Berg et al. [2021a] recently observed that operations that defer computation, such as evaluation strategies for λ application or *(multi-)staging* [Taha and Sheard, 2000], are beyond the expressiveness of scoped effects. Therefore, Van den Berg et al. [2021a] introduced another flavor of effects and handlers that they call *latent effects and handlers*.

In this paper we present a (surprisingly) simple alternative solution to the modularity problem with higher-order effects, which only uses standard effects and handlers and off-the-shelf generic programming techniques known from, e.g., *data types à la carte* [Swierstra, 2008].

3.1.3 *Solving the Modularity Problem: Elaboration Algebras*

We propose to define elaborations such as *censor* (Section 3.1.2) in a modular way. To this end, we introduce a new type of *computations with higher-order effects* which can be modularly elaborated into computations with only standard algebraic effects:

$$A \mathbin{!!} H \xrightarrow{\mathit{elaborate}} A \mathbin{!} \Delta \xrightarrow{\mathit{handle}} \mathit{Result}$$

Here $A \mathbin{!!} H$ is a computation type where A is a return type and H is a row comprising both algebraic and higher-order effects. The idea is that the higher-order effects in the row H are modularly elaborated into the row Δ. To achieve this, we define *elaborate* such that it can be modularly composed from separately defined elaboration cases, which we call elaboration *algebras* (for reasons we explain in Section 3.3). Using $A \mathbin{!!} H \Rrightarrow A \mathbin{!} \Delta$ as the type of elaboration algebras that elaborate the higher-order effects in H to Δ, we can modularly compose any pair of elaboration algebras

$e_1 : A \mathbin{!!} H_1 \Rrightarrow A \mathbin{!} \Delta$ and $e_2 : A \mathbin{!!} H_2 \Rrightarrow A \mathbin{!} \Delta$ into an algebra $e_{12} : A \mathbin{!!} H_1, H_2 \Rrightarrow A \mathbin{!} \Delta$.[35]

[35] *Readers familiar with data types à la carte [Swierstra, 2008] may recognize this as algebra composition.*

Elaboration algebras are as simple to define as non-modular elaborations such as *censor* (Section 3.1.2). For example, here is the elaboration algebra for the higher-order *Censor* effect whose only associated operation is the higher-order operation $\mathit{censor}_{op} : (\mathit{String} \to \mathit{String}) \to A \mathbin{!!} H \to A \mathbin{!!} H$:

$$\begin{aligned}
&\mathit{eCensor} : A \mathbin{!!} \mathit{Censor} \Rrightarrow A \mathbin{!} \mathit{Output}, \Delta \\
&\mathit{eCensor}\ (\mathit{censor}_{op}\ f\ m;\ k) = \mathbf{do}\ (x, s) \leftarrow (\mathbf{with}\ \mathit{hOut}\ \mathbf{handle}\ m); \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{out}\ (f\ s);\ k\ x
\end{aligned}$$

The implementation of *eCensor* is essentially the same as *censor*. There are two main differences. First, elaboration happens in-context, so the value yielded by the elaboration is passed to the context (or continuation) k. Second, and most importantly, programs that use the censor_{op} operation are now programmed against the interface given by *Censor*, meaning programs do not (and *cannot*) make assumptions about how censor_{op} is elaborated. As a consequence, we can modularly refine the elaboration of higher-order operations such as censor_{op}, without modifying the programs that use the operations. For example, the following program censors and replaces “Hello” with “Goodbye”:[36]

[36] *This program relies on the fact that it is generally possible to lift computation $A \mathbin{!} \Delta$ to $A \mathbin{!!} H$ when $\Delta \subseteq H$.*

$$\begin{aligned}
&\mathit{censorHello} : () \mathbin{!!} \mathit{Censor}, \mathit{Output} \\
&\mathit{censorHello} = \mathit{censor}_{op} \\
&\quad (\lambda s.\ \mathbf{if}\ (s \equiv \text{“Hello”})\ \mathbf{then}\ \text{“Goodbye”}\ \mathbf{else}\ s) \\
&\quad \mathit{hello}
\end{aligned}$$

Say we have a handler $\mathit{hOut}' : (\mathit{String} \to \mathit{String}) \to A \mathbin{!} \mathit{Output}, \Delta \Rightarrow (A \times \mathit{String}) \mathbin{!} \Delta$ which handles each operation $\mathit{out}\ s$ by applying a censor function $(\mathit{String} \to \mathit{String})$ to s before emitting it. Using this handler, we can give an alternative elaboration of censor_{op} which post-processes output strings *individually*:

$$\begin{aligned}
&\mathit{eCensor}' : A \mathbin{!!} \mathit{Censor} \Rrightarrow A \mathbin{!} \mathit{Output}, \Delta \\
&\mathit{eCensor}'\ (\mathit{censor}_{op}\ f\ m;\ k) = \mathbf{do}\ x \leftarrow (\mathbf{with}\ \mathit{hOut}'\ f\ \mathbf{handle}\ m); \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{out}\ s;\ k\ x
\end{aligned}$$

In contrast, *eCensor* applies the censoring function (*String* → *String*) to the batch output of the computation argument of a $censor_{op}$ operation. The batch output of *hello* is “Hello world!” which is unequal to “Hello”, so *eCensor* leaves the string unchanged. On the other hand, *eCensor′* censors the individually output “Hello”:

$$\textbf{with}\ \mathit{hOut}\ \textbf{handle}\ (\textbf{with}\ \mathsf{eCensor}\ \textbf{elaborate}\ \mathit{censorHello}) \\ \equiv ((), \text{“Hello world!”})$$

$$\textbf{with}\ \mathit{hOut}\ \textbf{handle}\ (\textbf{with}\ \mathsf{eCensor}'\ \textbf{elaborate}\ \mathit{censorHello}) \\ \equiv ((), \text{“Goodbye world!”})$$

Higher-order operations now have the same modularity benefits as algebraic operations.

3.1.4 *Contributions*

This paper formalizes the ideas sketched in this introduction by shallowly embedding them in Agda. However, the ideas transcend Agda. Similar shallow embeddings can be implemented in other dependently typed languages, such as Idris [Brady, 2013b]; but also in less dependently typed languages like Haskell, OCaml, or Scala.[37] By working in a dependently typed language we can state algebraic laws about interfaces of effectful operations, and prove that implementations of the interfaces respect the laws. We make the following technical contributions:

[37] *The artifact accompanying this paper [Van der Rest and Bach Poulsen, 2024] contains a shallow embedding of elaboration algebras in Haskell.*

- Section 5.2.3 describes how to encode algebraic effects in Agda, revisits the modularity problem with higher-order operations, and summarizes how scoped effects and handlers address the modularity problem, for some (*scoped* operations) but not all higher-order operations.
- Section 3.3 presents our solution to the modularity problem with higher-order operations. Our solution is to (1) type programs as *higher-order effect trees* (which we dub *hefty trees*), and (2) build modular elaboration algebras for

folding hefty trees into algebraic effect trees and handlers. The computations of type $A \mathbin{!!} H$ discussed in Section 3.1.3 correspond to hefty trees, and the elaborations of type $A \mathbin{!!} H \Rrightarrow A \mathbin{!} \Delta$ correspond to hefty algebras.

- Section 3.4 presents examples of how to define hefty algebras for common higher-order effects from the literature on effect handlers.

- Section 3.5 shows that hefty algebras support formal and modular reasoning on a par with algebraic effects and handlers, by developing reasoning infrastructure that supports verification of equational laws for higher-order effects such as exception catching. Crucially, proofs of correctness of elaborations are compositional. When composing two proven correct elaboration, correctness of the combined elaboration follows immediately without requiring further proof work.

Section 5.6 discusses related work and Section 5.7 concludes. An artifact containing the code of the paper and a Haskell embedding of the same ideas is available online [Van der Rest and Bach Poulsen, 2024]. A subset of the contributions of this paper were previously published in a conference paper [Bach Poulsen and Van der Rest, 2023]. While that version of the paper too discusses reasoning about higher-order effects, the correctness proofs were non-modular, in that they make assumptions about the order in which the algebraic effects implementing a higher-order effect are handled. When combining elaborations, these assumptions are often incompatible, meaning that correctness proofs for the individual elaborations do not transfer to the combined elaboration. As a result, one would have to re-prove correctness for every combination of elaborations. For this extended version, we developed reasoning infrastructure to support modular reasoning about higher-order effects in Section 3.5, and proved that correctness of elaborations is preserved under composition of elaborations.

3.2 ALGEBRAIC EFFECTS AND HANDLERS IN AGDA

This section describes how to encode algebraic effects and handlers in Agda. We do not assume familiarity with Agda and explain Agda specific notation in sidenotes. Sections 3.2.1 to 3.2.4 defines algebraic effects and handlers; Section 3.2.5 revisits the problem of defining higher-order effects using algebraic effects and handlers; and Section 3.2.6 discusses how scoped effects [Wu et al., 2014, Piróg et al., 2018, Yang et al., 2022] solves the problem for *scoped* operations but not all higher-order operations.

3.2.1 *Algebraic Effects and The Free Monad*

We encode algebraic effects in Agda by representing computations as an abstract syntax tree given by the *free monad* over an *effect signature*. Such effect signatures are traditionally [Awodey, 2010, Swierstra, 2008, Kiselyov and Ishii, 2015, Wu et al., 2014, Kammar et al., 2013] given by a *functor*; i.e., a type of kind Set → Set together with a (lawful) mapping function.[38] In our Agda implementation, effect signature functors are defined by giving a *container* [Abbott et al., 2003, 2005a]. Each container corresponds to a value of type Set → Set that is both *strictly positive* and *universe consistent* [Martin-Löf, 1984], meaning they are a constructive approximation of endofunctors on Set. Effect signatures are given by a (dependent) record type:

[38] *Set is the type of types in Agda. More generally, functors mediate between different* categories. *For simplicity, this paper only considers* endofunctors *on Set, where an endofunctor is a functor whose domain and codomain coincides; e.g., Set → Set.*

```
record Effect : Set₁ where
  field Op  : Set
        Ret : Op → Set
```

Here, Op is the set of operations, and Ret defines the *return type* for each operation in the set Op. The extension of an effect signature, ⟦_⟧, reflects its input of type Effect as a value of type Set → Set:

```
⟦_⟧ : Effect → Set → Set
⟦ Δ ⟧ X = Σ (Op Δ) λ op → Ret Δ op → X
```

The extension of an effect Δ into Set → Set is indeed a functor, as witnessed by the following function:[39]

[39] *To show that this is truly a functor, we should also prove that map-sig satisfies the* functor laws. *We will not make use of these functor laws in this paper, so we omit them.*

```
map-sig : (X → Y) → ⟦ Δ ⟧ X → ⟦ Δ ⟧ Y
map-sig f (op , k) = ( op , f ∘ k )
```

As discussed in the introduction, computations may use multiple different effects. Effect signatures are closed under co-products:[40] [41]

```
_⊕_ : Effect → Effect → Effect
Op  (Δ₁ ⊕ Δ₂) = Op Δ₁ ⊎ Op Δ₂
Ret (Δ₁ ⊕ Δ₂) = [ Ret Δ₁ , Ret Δ₂ ]
```

We compute the co-product of two effect signatures by taking the disjoint sum of their operations and combining the return type mappings pointwise. We co-products to encode effect rows. For example, The effect $\Delta_1 \oplus \Delta_2$ corresponds to the row union denoted as Δ_1, Δ_2 in the introduction.

The syntax of computations with effects Δ is given by the free monad over Δ. We encode the free monad as follows:

```
data Free (Δ : Effect) (A : Set) : Set where
  pure   : A                  → Free Δ A
  impure : ⟦ Δ ⟧ (Free Δ A) → Free Δ A
```

Here, pure is a computation with no side effects, whereas impure is an operation whose syntax is given by the functor ⟦ Δ ⟧. By applying this functor to Free Δ A, we encode an operation whose *continuation* may contain more effectful operations.[42] To see in what sense, let us consider an example.

EXAMPLE. The following data type defines an operation for outputting a string. Below it is its corresponding effect signature.

```
data OutOp : Set where
  out : String → OutOp

Output : Effect
Op  Output         = OutOp
Ret Output (out s) = ⊤
```

[40] *The* _⊕_ *function uses* copattern matching: https://agda.readthedocs.io/en/v2.6.2.2/language/copatterns.html. *The* Op *line defines how to compute the* Op *field of the record produced by the function; and similarly for the* Ret *line.*

[41] _⊎_ *is a* disjoint sum *type from the Agda standard library. It has two constructors,* $inj_1 : A \to A \uplus B$ *and* $inj_2 : B \to A \uplus B$. *The* [_,_] *function (also from the Agda standard library) is the* eliminator *for the disjoint sum type. Its type is* [_,_] : $(A \to X) \to (B \to X) \to (A \uplus B) \to X$.

[42] *By unfolding the definition of* ⟦_⟧ *one can see that our definition of the free monad is identical to the I/O trees of* Hancock and Setzer [2000], *or the so-called* freer monad *of* Kiselyov and Ishii [2015].

The effect signature says that out returns a unit value (⊤ is the unit type). Using this, we can write a simple hello world corresponding to the *hello* program from Section 3.1:

```
hello : Free Output ⊤
hello = impure
  ( out "Hello"
  , λ _ → impure (out " world!" , λ x → pure x))
```

Section 3.2.1 shows how to make this program more readable by using monadic do notation.

The hello program above makes use of just a single effect. Say we want to use another effect, Throw, with a single operation, throw, which represents throwing an exception (therefore having the empty type ⊥ as its return type):

```
data ThrowOp : Set where
  throw : ThrowOp

Throw : Effect
Op  Throw = ThrowOp
Ret Throw throw = ⊥
```

Programs that use multiple effects, such as Output and Throw, are unnecessarily verbose. For example, consider the following program which prints two strings before throwing an exception:[43]

[43] ⊥-elim *is the eliminator for the empty type, encoding the* principle of explosion*:* ⊥-elim : ⊥ → *A*.

```
hello-throw : Free (Output ⊕ Throw) A
hello-throw = impure (inj₁ (out "Hello") , λ _ →
                impure (inj₁ (out " world!") , λ _ →
                  impure (inj₂ throw , ⊥-elim)))
```

To reduce syntactic overhead, we use *row insertions* and *smart constructors* [Swierstra, 2008].

3.2.2 *Row Insertions and Smart Constructors*

A *smart constructor* constructs an effectful computation comprising a single operation. The type of this computation is polymorphic in what other effects the computation has. For example, the type of a smart constructor for the out effect is:

```
‵out : ⦃ Output ≲ Δ ⦄ → String → Free Δ ⊤
```

Here, the ⦃ Output $\lesssim$ Δ ⦄ type declares the row insertion witness as an *instance argument* of ‵out. Instance arguments in Agda are conceptually similar to type class constraints in Haskell: when we call ‵out, Agda will attempt to automatically find a witness of the right type, and implicitly pass this as an argument [Devriese and Piessens, 2011]. Thus, calling ‵out will automatically inject the Output effect into some larger effect row Δ.

We define the $\lesssim$ order on effect rows in terms of a different $\Delta_1 \bullet \Delta_2 \approx \Delta$ which witnesses that any operation of Δ is isomorphic to *either* an operation of Δ_1 *or* an operation of Δ_2:[44]

```
record _•_≈_ (Δ₁ Δ₂ Δ : Effect) : Set₁ where
  field reorder : ∀ {X} → ⟦ Δ₁ ⊕ Δ₂ ⟧ X ↔ ⟦ Δ ⟧ X
```

Using this, the $\lesssim$ order is defined as follows:

```
_≲_ : (Δ₁ Δ₂ : Effect) → Set₁
Δ₁ ≲ Δ₂ = ∃ λ Δ′ → Δ₁ • Δ′ ≈ Δ₂
```

It is straightforward to show that $\lesssim$ is a *preorder*; i.e., that it is a *reflexive* and *transitive* relation.

We can also define the following function, which uses a $\Delta_1 \lesssim \Delta_2$ witness to coerce an operation of effect type Δ_1 into an operation of some larger effect type Δ_2.[45]

```
inj : ⦃ Δ₁ ≲ Δ₂ ⦄ → ⟦ Δ₁ ⟧ A → ⟦ Δ₂ ⟧ A
inj ⦃ _ , w ⦄ (c , k) = w .reorder .to (inj₁ c , k)
```

Furthermore, we can freely coerce the operations of a computation from one effect row type to a different effect row type:[46] [47]

```
hmap-free : ∀[ ⟦ Δ₁ ⟧ ⇒ ⟦ Δ₂ ⟧ ] → ∀[ Free Δ₁ ⇒ Free Δ₂ ]
hmap-free θ (pure x)       = pure x
hmap-free θ (impure (c , k)) = impure (θ (c , hmap-free θ ∘ k))
```

Using this infrastructure, we can now implement a generic inject function which lets us define smart constructors for operations such as the out operation discussed in the previous subsection.

44 *Here* ↔ *is the type of an* isomorphism *on* Set *from the Agda Standard Library. It is given by a record with two fields: the* to *field represents the* → *direction of the isomorphism, and* from *field represents the* ← *direction of the isomorphism.*

45 *The dot notation* w .reorder *projects the* reorder *field of the record* w.

46 *The notation* ∀[_] *is from the Agda Standard library, and is defined as follows:* $\forall[\, P\,] = \forall\, x \to P\, x$.

47 *We can think of the* hmap-free *function as a "higher-order" map for* Free*: given a natural transformation between (the extension of) signatures, we can can transform the signature of a computation. This amounts to the observation that* Free *is a functor over the category of containers and container morphisms; assuming* hmap-free *preserves naturality.*

```
inject : { Δ₁ ≲ Δ₂ } → Free Δ₁ A → Free Δ₂ A
inject = hmap-free inj

‵out : { Output ≲ Δ } → String → Free Δ ⊤
‵out s = inject (impure (out s , pure))
```

3.2.3 *Fold and Monadic Bind for Free*

Since Free Δ is a monad, we can sequence computations using *monadic bind*, which is naturally defined in terms of the fold over Free.

```
fold : (A → B) → Alg Δ B → Free Δ A → B
fold g a (pure x) = g x
fold g a (impure (op , k)) = a (op , fold g a ∘ k)


Alg : (Δ : Effect) (A : Set) → Set
Alg Δ A = ⟦ Δ ⟧ A → A
```

Besides the input computation to be folded (last parameter), the fold is parameterized by a function $A \to B$ (first parameter) which folds a pure computation, and an *algebra* Alg Δ A (second parameter) which folds an impure computation. We call the latter an algebra because it corresponds to an F-algebra [Arbib and Manes, 1975, Pierce, 1991] over the signature functor of Δ, denoted F_Δ. That is, a tuple (A, α) where A is an object called the *carrier* of the algebra, and α a morphism $F_\Delta(A) \to A$. Using fold, monadic bind for the free monad is defined as follows:

```
_≫=_ : Free Δ A → (A → Free Δ B) → Free Δ B
m ≫= g = fold g impure m
```

Intuitively, m ≫= g concatenates g to all the leaves in the computation m.

EXAMPLE. The following defines a smart constructor for throw:

```
‵throw : { Throw ≲ Δ } → Free Δ A
```

Using this and the definition of ≫= above, we can use **do**-notation in Agda to make the hello-throw program from Section 3.2.1 more readable:

```
hello-throw₁ : { Output ≲ Δ } → { Throw ≲ Δ } → Free Δ A
hello-throw₁ = do `out "Hello"; `out " world!"; `throw
```

This illustrates how we use the free monad to write effectful programs against an interface given by an effect signature. Next, we define *effect handlers*.

3.2.4 *Effect Handlers*

An effect handler implements the interface given by an effect signature, interpreting the syntactic operations associated with an effect. Like monadic bind, effect handlers can be defined as a fold over the free monad. The following type of *parameterized handlers* defines how to fold respectively pure and impure computations:

```
record ⟨_!_⇒_⇒_!_⟩ (A : Set) (Δ : Effect)
                   (P : Set) (B : Set)
                   (Δ′ : Effect) : Set₁ where
  field ret : A → P → Free Δ′ B
        hdl : Alg Δ (P → Free Δ′ B)
```

A handler of type ⟨ A ! Δ ⇒ P ⇒ B ! Δ′ ⟩ is parameterized in the sense that it turns a computation of type Free Δ A into a parameterized computation of type P → Free Δ′ B. The following function does so by folding using ret, hdl, and a to-front function:

```
to-front : { Δ₁ • Δ₂ ≈ Δ } → Free Δ A → Free (Δ₁ ⊕ Δ₂) A
to-front { w } = hmap-free (w .reorder .from)

given_handle_ : { w : Δ₁ • Δ₂ ≈ Δ }
              → ⟨ A ! Δ₁ ⇒ P ⇒ B ! Δ₂ ⟩
              → Free Δ A → (P → Free Δ₂ B)
given_handle_ h m = fold
  (ret h)
  ( λ where (inj₁ c , k) p → hdl h (c , k) p
            (inj₂ c , k) p → impure (c , flip k p) )
    (to-front m)
```

Comparing with the syntax we used to explain algebraic effects and handlers in the introduction, the ret field corresponds to the **return** case of the handlers from the introduction, and hdl corresponds to the cases that define how operations are handled. The parameterized handler type ⟨ A ! Δ ⇒ P ⇒ B ! Δ′ ⟩ corresponds to the type $A \,!\, \Delta, \Delta' \Rightarrow P \to B \,!\, \Delta'$, and given h handle m corresponds to **with** h **handle** m.

Using this type of handler, the *hOut* handler from the introduction can be defined as follows:

```
hOut : ⟨ A ! Output ⇒ ⊤ ⇒ (A × String) ! Δ ⟩
ret hOut x _ = pure (x , "")
hdl hOut (out s , k) p = do (x , s′) ← k tt p; pure (x , s ++ s′)
```

The handler *hOut* in Section 3.1.1 did not bind any parameters. However, since we are encoding it as a *parameterized* handler, hOut now binds a unit-typed parameter. Besides this difference, the handler is the same as in Section 3.1.1. We can use the hOut handler to run computations. To this end, we introduce a Nil effect with no associated operations which we will use to indicate where an effect row ends:

```
Nil : Effect
Op  Nil = ⊥
Ret Nil = ⊥-elim


un : Free Nil A → A
un (pure x) = x
```

Using these, we can run a simple hello world program:[48]

[48] *The* refl *constructor is from the Agda standard library, and witnesses that a propositional equality (≡) holds.*

```
hello′ : ⦃ Output ≲ Δ ⦄ → Free Δ ⊤
hello′ = do
  ‵out "Hello"; ‵out " world!"

test-hello : un (given hOut handle hello′ $ tt)
```

```
data StateOp : Set where                State : Effect
  get :          StateOp                Op State = StateOp
  put : ℕ → StateOp                     Ret State get      = ℕ
                                        Ret State (put n) = ⊤
```

```
hSt : ⟨ A ! State ⇒ ℕ ⇒ (A × ℕ) ! Δ′ ⟩
ret hSt x s = pure (x , s)
hdl hSt (put m , k) n = k tt m
hdl hSt (get , k) n = k n     n

‵incr : ⦃ State ≲ Δ ⦄ → Free Δ ⊤
‵incr = do n ← ‵get; ‵put (n + 1)

incr-test : un ((given hSt handle ‵incr) 0) ≡ (tt , 1)
incr-test = refl
```

Figure 8: A state effect (upper), its handler (hSt below), and a simple test (incr-test, also below) which uses (the elided) smart constructors for get and put

```
            ≡ (tt , "Hello world!")
test-hello = refl
```

An example of parameterized (as opposed to unparameterized) handlers, is the state effect. Figure 8 declares and illustrates how to handle such an effect with operations for reading (get) and changing (put) the state of a memory cell holding a natural number.

3.2.5 *The Modularity Problem with Higher-Order Effects, Revisited*

Section 3.1.2 described the modularity problem with higher-order effects, using a higher-order operation that interacts with output as an example. In this section we revisit the problem, framing it in terms of the definitions introduced in the previous section. To this end, we use a different effect whose interface is summarized by the CatchM record below. The record asserts that the computation type M : Set → Set has at least a higher-order operation catch and a first-order operation throw:

```
record CatchM (M : Set → Set) : Set₁ where
  field catch : M A → M A → M A
        throw :               M A
```

The idea is that throw throws an exception, and catch m_1 m_2 handles any exception thrown during evaluation of m_1 by running m_2 instead. The problem is that we cannot give a modular definition of operations such as catch using algebraic effects and handlers alone. As discussed in Section 3.1.2, the crux of the problem is that algebraic effects and handlers provide limited support for higher-order operations. However, as also discussed in Section 3.1.2, we can encode catch in terms of more primitive effects and handlers, such as the following handler for the Throw effect:

```
hThrow : ⟨ A ! Throw ⇒ ⊤ ⇒ (Maybe A) ! Δ′ ⟩
ret hThrow x _ = pure (just x)
hdl hThrow (throw , k) _ = pure nothing
```

The handler modifies the return type of the computation by decorating it with a Maybe. If no exception is thrown, ret wraps the yielded value in a just constructor. If an exception is thrown, the handler never invokes the continuation k and aborts the computation by returning nothing instead. We can elaborate catch into an inline application of hThrow. To do so we make use of *effect masking* which lets us "weaken" the type of a computation by inserting extra effects in an effect row:

```
♯_ : { Δ₁ ≲ Δ₂ } → Free Δ₁ A → Free Δ₂ A
```

Using this, the following elaboration defines a semantics for the catch operation:[49] [50]

```
catch : { Throw ≲ Δ } → Free Δ A → Free Δ A → Free Δ A
catch m₁ m₂
  = (♯ (given hThrow handle m₁) tt) ≫= maybe pure m₂
```

If m_1 does not throw an exception, we return the produced value. If it does, m_2 is run.

49 *The maybe function is the eliminator for the Maybe type. Its first parameter is for eliminating a just; the second for nothing. Its type is maybe : (A → B) → B → Maybe A → B.*

50 *The instance resolution machinery of Agda requires some help to resolve the instance argument of ♯ here. We provide a hint to Agda's instance resolution machinery in an implicit instance argument that we omit for readability in the paper. In the rest of this paper, we will occasionally follow the same convention.*

As observed by Wu et al. [2014], programs that use elaborations such as catch are less modular than programs that only use plain algebraic operations. In particular, the effect row type of computations no longer represents the interface of operations that we use to write programs, since the catch elaboration is not represented in the effect type at all. So we have to rely on different machinery if we want to refactor, optimize, or change the semantics of catch without having to change programs that use it.

In the next subsection we describe how to define effectful operations such as catch modularly using scoped effects and handlers, and discuss how this is not possible for, e.g., operations representing λ-abstraction.

3.2.6 *Scoped Effects and Handlers*

This subsection gives an overview of scoped effects and handlers. While the rest of the paper can be read and understood without a deep understanding of scoped effects and handlers, we include this overview to facilitate comparison with the alternative solution that we introduce in Section 3.3.

Scoped effects extend the expressiveness of algebraic effects to support a class of higher-order operations that Wu et al. [2014], Piróg et al. [2018], Yang et al. [2022] call *scoped operations*. We illustrate how scoped effects work, using a freer monad encoding of the endofunctor algebra approach of Yang et al. [2022]. The work of Yang et al. [2022] does not include examples of modular handlers, but the original paper on scoped effects and handlers by Wu et al. [2014] does. We describe an adaptation of the modular handling techniques due to Wu et al. [2014] to the endofunctor algebra approach of Yang et al. [2022].

3.2.6.1 *Scoped Programs*

Scoped effects extend the free monad data type with an additional row for scoped operations. The return and call constructors of Prog below correspond to the pure and impure constructors of the free monad, whereas enter is new:

```
data Prog (Δ γ : Effect) (A : Set) : Set where
  return : A                              → Prog Δ γ A
  call   : ⟦ Δ ⟧ (Prog Δ γ A)             → Prog Δ γ A
  enter  : ⟦ γ ⟧ (Prog Δ γ (Prog Δ γ A))  → Prog Δ γ A
```

Here, the enter constructor represents a higher-order operation with *sub-scopes*; i.e., computations that themselves return computations:

$$\underbrace{\mathsf{Prog}\ \Delta\ \gamma}_{\text{outer}}\ (\underbrace{\mathsf{Prog}\ \Delta\ \gamma}_{\text{inner}}\ A)$$

This type represents *scoped* computations in the sense that outer computations can be handled independently of inner ones, as we illustrate in Section 3.2.6.2. One way to think of inner computations is as continuations (or join-points) of sub-scopes.

Using Prog, the catch operation can be defined as a scoped operation:

```
data CatchOp : Set where
  catch : CatchOp

Catch : Effect
Op  Catch = CatchOp
Ret Catch catch = Bool
```

The effect signature indicates that Catch has two scopes since Bool has two inhabitants. Following Yang et al. [2022], scoped operations are handled using a structure-preserving fold over Prog:

```
hcata :   (∀ {X} → X → G X)
        → CallAlg   Δ G
        → EnterAlg γ G
        → Prog Δ γ A → G A

CallAlg : (Δ : Effect) (G : Set → Set) → Set₁
CallAlg Δ G =
  {A : Set} → ⟦ Δ ⟧ (G A) → G A

EnterAlg : (γ : Effect) (G : Set → Set) → Set₁
EnterAlg γ G =
  {A B : Set} → ⟦ γ ⟧ (G (G A)) → G A
```

The first argument represents the case where we are folding a return node; the second and third correspond to respectively call and enter.

3.2.6.2 *Scoped Effect Handlers*

The following defines a type of parameterized scoped effect handlers:

```
record ⟨●!_!_⇒_⇒_●!_!_⟩ (Δ γ : Effect) (P : Set)
                        (G : Set → Set)
                        (Δ′ γ′ : Effect) : Set₁ where
  field ret    :  X → P → Prog Δ′ γ′ (G X)
        hcall  :  CallAlg   Δ (λ X → P → Prog Δ′ γ′ (G X))
        henter :  EnterAlg γ (λ X → P → Prog Δ′ γ′ (G X))
        glue   :  (k : C → P → Prog Δ′ γ′ (G X)) (r : G C) → P
               → Prog Δ′ γ′ (G X)
```

A handler of type ⟨● ! Δ ! γ ⇒ P ⇒ G ●! Δ′ ! γ ⟩ handles operations of Δ and γ *simultaneously* and turns a computation Prog Δ γ A into a parameterized computation of type $P \to$ Prog $\Delta'\ \gamma'\ (G\ A)$. The ret and hcall cases are similar to the ret and hdl cases from Section 3.2.4. The crucial addition which adds support for higher-order operations is the henter case.

The henter field is given by an EnterAlg case. This case takes as input a scoped operation whose outer and inner computation have been folded into a parameterized computation of type $P \to$ Prog $\Delta'\ \gamma'\ (G\ X)$; and returns as output an interpretation of that operation as a computation of type $P \to$ Prog $\Delta'\ \gamma'\ (G\ X)$. The glue function is used for modularly *weaving* [Wu et al., 2014] side effects of handlers through sub-scopes of yet-unhandled operations.

3.2.6.3 *Weaving*

To see why glue is needed, it is instructional to look at how the fields in the record type above are used to fold over Prog:

```
given_handle-scoped_ :  { w₁ : Δ₁ • Δ₂ ≈ Δ }
                        { w₂ : γ₁ • γ₂ ≈ γ }
                      → ⟨•! Δ₁ ! γ₁ ⇒ P ⇒ G •! Δ₂ ! γ₂ ⟩
                      → Prog Δ γ A → P → Prog Δ₂ γ₂ (G A)
given h handle-scoped m = hcata (ret h)
  ⊕[ hcall h
   , (λ where (c , k) p → call (c , flip k p)) ]
  ⊕[ (λ {A} → henter h {A})
   , (λ where
         (c , k) p → enter (c ,
           λ x → map-prog (λ y → glue h id y p) (k x p))
   ) ]′
   (to-frontΔ (to-frontγ m))
```

The second to last line above shows how glue is used. Because hcata eagerly folds the current handler over scopes (*sc*), there is a mismatch between the type that the continuation expects (B) and the type that the scoped computation returns ($G\ B$). The glue function fixes this mismatch for the particular return type modification G : Set → Set of a parameterized scoped effect handler.

The scoped effect handler for exception catching is thus:[51]

[51] *Here, flip : (A → B → C) → (B → A → C).*

```
hCatch : ⟨•! Throw ! Catch ⇒ ⊤ ⇒ Maybe •! Δ ! γ ⟩
ret    hCatch x _ = return (just x)
hcall  hCatch (throw , k) _ = return nothing
henter hCatch (catch , k) _ = k true tt ≫= λ where
  (just f) → f tt
  nothing → k false tt ≫= maybe (_$ tt) (return nothing)
glue hCatch k x _ = maybe (flip k tt) (return nothing) x
```

The henter field for the catch operation first runs m_1. If no exception is thrown, the value produced by m_1 is forwarded to k. Otherwise, m_2 is run and its value is forwarded to k, or its exception is propagated. The glue field of hCatch says that, if an unhandled exception is thrown during evaluation of a scope, the continuation is discarded and the exception is propagated; and if no exception is thrown the continuation proceeds normally.

3.2.6.4 *Discussion and Limitations*

As observed by Van den Berg et al. [2021a], some higher-order effects do not correspond to scoped operations. In particular, the LambdaM record shown below is not a scoped operation:

```
record LambdaM (V : Set) (M : Set → Set) : Set₁ where
  field lam : (V → M V) → M V
        app : V → M V    → M V
```

The lam field represents an operation that constructs a λ value. The app field represents an operation that will apply the function value in the first parameter position to the argument computation in the second parameter position. The app operation has a computation as its second parameter so that it remains compatible with different evaluation strategies.

To see why the operations summarized by the LambdaM record above are not scoped operations, let us revisit the enter constructor of Prog:

$$\mathsf{enter} : [\![\, \gamma \,]\!] \; (\underbrace{\mathsf{Prog}\ \Delta\ \gamma}_{\text{outer}} \; (\underbrace{\mathsf{Prog}\ \Delta\ \gamma}_{\text{inner}}\ A)) \to \mathsf{Prog}\ \Delta\ \gamma\ A$$

As summarized earlier in this subsection, enter lets us represent higher-order operations (specifically, *scoped operations*), whereas call does not (only *algebraic operations*). Just like we defined the computational parameters as scopes (given by the outer Prog in the type of enter), we might try to define the body of a lambda as a scope in a similar way. However, whereas the catch operation always passes control to its continuation (the inner Prog), the lam effect is supposed to package the body of the lambda into a value and pass this value to the continuation (the inner computation). Because the inner computation is nested within the outer computation, *the only way to gain access to the inner computation (the* *continuation) is by first running the outer computation (the body of the lambda)*. This does not give us the right semantics.

It is possible to elaborate the LambdaM operations into more primitive effects and handlers, but as discussed in Sections 3.1.2 and 3.2.5, such elaborations are not modular. In the next section we show how to make such elaborations modular.

3.3 HEFTY TREES AND ALGEBRAS

As observed in Section 3.2.5, operations such as catch can be elaborated into more primitive effects and handlers. However, these elaborations are not modular. We solve this problem by factoring elaborations into interfaces of their own to make them modular.

To this end, we first introduce a new type of abstract syntax trees (Sections 3.3.1 to 3.3.3) representing computations with higher-order operations, which we dub *hefty trees* (an acronymic pun on *h*igher-order *ef*fects). We then define elaborations as algebras (*hefty algebras*; Section 3.3.4) over these trees. The following pipeline summarizes the idea, where H is a *higher-order effect signature*:

$$\mathsf{Hefty}\ H\ A \xrightarrow{elaborate} \mathsf{Free}\ \Delta\ A \xrightarrow{handle} Result$$

For the categorically inclined reader, Hefty conceptually corresponds to the initial algebra of the functor *HeftyF* $\mathsf{H}\ \mathsf{A}\ \mathsf{R} = \mathsf{A} + \mathsf{H}\ \mathsf{R}\ (\mathsf{R}\ \mathsf{A})$ where $\mathsf{H} : (\mathsf{Set} \to \mathsf{Set}) \to (\mathsf{Set} \to \mathsf{Set})$ defines the signature of higher-order operations and is a *higher-order functor*, meaning we have both the usual functorial *map* : $(\mathsf{X} \to \mathsf{Y}) \to \mathsf{H}\ \mathsf{F}\ \mathsf{X} \to \mathsf{H}\ \mathsf{F}\ \mathsf{Y}$ for any functor F as well as a function *hmap* : $\mathrm{Nat}(\mathsf{F},\mathsf{G}) \to \mathrm{Nat}(\mathsf{H}\ \mathsf{F},\mathsf{H}\ \mathsf{G})$ which lifts natural transformations between any F and G to a natural transformation between $\mathsf{H}\ \mathsf{F}$ and $\mathsf{H}\ \mathsf{G}$. A hefty algebra is then an F-algebra over a higher-order signature functor H. The notion of elaboration that we introduce in Section 3.3.4 is an F-algebra whose carrier is a "first-order" effect tree (Free Δ).

In this section, we encode this conceptual framework in Agda using a container-inspired approach to represent higher-order signature functors H as a strictly positive type. We discuss and compare our approach with previous work in Section 3.3.5.

3.3.1 *Generalizing Free to Support Higher-Order Operations*

As summarized in Section 3.2.1, Free Δ A is the type of abstract syntax trees representing computations over the effect signature Δ. Our objective is to arrive at a more general type of abstract syntax trees representing computations involving (possibly) higher-order operations. To realize this objective, let us consider how to syntactically represent this variant of the *censor* operation (Section 3.1.2), where M is the type of abstract syntax trees whose type we wish to define:

$$\mathsf{censor}_{\mathsf{op}} : (\mathsf{String} \to \mathsf{String}) \to M \top \to M \top$$

We call the second parameter of this operation a *computation parameter*. Using Free, computation parameters can only be encoded as continuations. But the computation parameter of $\mathsf{censor}_{\mathsf{op}}$ is *not* a continuation, since

$$\mathsf{do}\ (\mathsf{censor}_{\mathsf{op}}\ f\ m);\ \grave{}\mathsf{out}\ s \quad \not\equiv \quad \mathsf{censor}_{\mathsf{op}}\ f\ (\mathsf{do}\ m;\ \grave{}\mathsf{out}\ s).$$

The crux of the issue is how signature functors $[\![\ \Delta\ ]\!] : \mathsf{Set} \to \mathsf{Set}$ are defined. Since this is an endofunctor on Set, the only suitable option in the impure constructor is to apply the functor to the type of *continuations*:

$$\mathsf{impure} : [\![\ \Delta\ ]\!]\ (\ \underbrace{\mathsf{Free}\ \Delta\ A}_{\text{continuation}}\ ) \to \mathsf{Free}\ \Delta\ A$$

A more flexible approach would be to allow signature functors to build computation trees with an *arbitrary return type*, including the return type of the continuation. A *higher-order* signature functor of some higher-order signature H, written $[\![\ H\ ]\!]^{\mathsf{H}} : (\mathsf{Set} \to \mathsf{Set}) \to \mathsf{Set} \to \mathsf{Set}$, would fit that bill. Using such a signature functor, we could define the impure case as follows:

$$\mathsf{impure} : [\![\, H \,]\!]^{\mathsf{H}} \, (\underbrace{\mathsf{Hefty}\ H}_{\text{computation type}}) \overbrace{A}^{\text{continuation return type}} \to \mathsf{Hefty}\ H\ A$$

Here, Hefty is the type of the free monad using higher-order signature functors instead. In the rest of this subsection, we consider how to define higher-order signature functors H, their higher-order functor extensions $[\![\_]\!]^{\mathsf{H}}$, and the type of Hefty trees.

Recall how we defined plain algebraic effects in terms of *containers*:

```
record Effect : Set₁ where
  field Op  : Set
        Ret : Op → Set
```

Here, Op is the type of operations, and Ret defines the return type of each operation. In order to allow operations to have sub-computations, we generalize this type to allow each operation to be associated with a number of sub-computations, where each sub-computation can have a different return type. The following record provides this generalization:

```
record Effectᴴ : Set₁ where
  field Opᴴ  : Set                                  -- As before
        Retᴴ : Opᴴ → Set                            -- As before
        Fork : Opᴴ → Set                            -- New
        Ty   : {op : Opᴴ} (ψ : Fork op) → Set       -- New
```

The set of operations is still given by a type field (Op^{H}), and each operation still has a return type ($\mathsf{Ret}^{\mathsf{H}}$). Fork associates each operation with a type that indicates how many sub-computations (or *forks*) an operation has, and Ty indicates the return type of each such fork. For example, say we want to encode an operation *op* with two sub-computations with different return types, and whose return type is of a unit type. That is, using M as our type of computations:

$$op : M\ \mathbb{Z} \to M\ \mathbb{N} \to M\ \top$$

The following signature declares a higher-order effect signature with a single operation with return type $\top$, and with two forks (we use Bool to encode this fact), and where each fork has, respectively $\mathbb{Z}$ and $\mathbb{N}$ as return types.

```
example-op : Effectᴴ
Opᴴ  example-op       = ⊤    -- A single operation
Retᴴ example-op tt    = ⊤    -- with return type ⊤
Fork example-op tt    = Bool -- with two forks
Ty   example-op false = ℤ    -- one fork has type ℤ
Ty   example-op true  = ℕ    -- the other has type ℕ
```

The extension of higher-order effect signatures implements the intuition explained above:

```
⟦_⟧ᴴ : Effectᴴ → (Set → Set) → Set → Set
⟦ H ⟧ᴴ M X = Σ (Opᴴ H) λ op →
  (Retᴴ H op → M X) × ((ψ : Fork H op) → M (Ty H ψ))
```

Let us unpack this definition.

$$\underbrace{\Sigma\ (\ \mathsf{Op}^{\mathsf{H}}\ H)\ \lambda\ op \to}_{(1)}$$
$$(\underbrace{\mathsf{Ret}^{\mathsf{H}}\ H\ op \to M\ X}_{(2)}) \times ((\underbrace{\psi : \mathsf{Fork}\ H\ op}_{(3)}) \to \underbrace{M\ (\mathsf{Ty}\ H\ \psi)}_{(4)})$$

The extension of a higher-order signature functor is given by (1) the sum of operations of the signature, where each operation has (2) a continuation (of type $M\ X$) that expects to be passed a value of the operation's return type, and (3) a set of forks where each fork is (4) a computation that returns the expected type for each fork.

Using the higher-order signature functor and its extension above, our generalized free monad becomes:

```
data Hefty (H : Effectᴴ) (A : Set) : Set where
  pure   : A → Hefty H A
  impure : ⟦ H ⟧ᴴ (Hefty H) A → Hefty H A
```

This type of Hefty trees can be used to define higher-order operations with an arbitrary number of computation parameters, with arbitrary return types. Using this type, and using a co-product for higher-order effect signatures (_∔_) which is analogous to the co-product for algebraic effect signatures in Section 3.2.2, Figure 9 represents the syntax of the censor_{op} operation.

```
                                        Censor : Effectᴴ
data CensorOp : Set where               Opᴴ  Censor                = CensorOp
  censor :   (String → String)          Retᴴ Censor (censor f)     = ⊤
           → CensorOp                   Fork Censor (censor f)     = ⊤
                                        Ty   Censor {censor f} tt  = ⊤
```

```
censorₒₚ : (String → String) → Hefty (Censor ∔ H) ⊤ → Hefty (Censor ∔ H) ⊤
censorₒₚ f m = impure (inj₁ (censor f) , (λ where tt → m) , pure)
```

Figure 9: A higher-order censor effect and operation, with a single computation parameter (declared with Op = ⊤ in the effect signature top right) with return type ⊤ (declared with Ret = λ _ → ⊤ top right)

Just like Free, Hefty trees can be sequenced using monadic bind. Unlike for Free, the monadic bind of Hefty is not expressible in terms of the standard fold over Hefty trees. The difference between Free and Hefty is that Free is a regular data type whereas Hefty is a *nested datatype* [Bird and Paterson, 1999]. The fold of a nested data type is limited to describe *natural transformations*. As Bird and Paterson [1999] show, this limitation can be overcome by using a *generalized fold*, but for the purpose of this paper it suffices to define monadic bind as a recursive function:

```
_≫=_ : Hefty H A → (A → Hefty H B) → Hefty H B
pure x            ≫= g = g x
impure (op , k , s) ≫= g = impure (op , (_≫= g) ∘ k , s)
```

The bind behaves similarly to the bind for Free; i.e., $m \gg\!= g$ concatenates g to all the leaves in the continuations (but not computation parameters) of m.

In Section 3.3.4 we show how to modularly elaborate higher-order operations into more primitive algebraic effects and handlers (i.e., computations over Free), by folding modular elaboration algebras (*hefty algebras*) over Hefty trees. First, we show (in Section 3.3.2) how Hefty trees support programming against an interface of both algebraic and higher-order operations. We also address (in Section 3.3.3) the question of how to encode effect signatures for higher-order operations whose computation parameters have polymorphic return types, such as the highlighted A below:

$$\text{`catch} : \text{Hefty}\ H\ \boxed{A} \to \text{Hefty}\ H\ \boxed{A} \to \text{Hefty}\ H\ \boxed{A}$$

3.3.2 *Programs with Algebraic and Higher-Order Effects*

Any algebraic effect can be lifted to a higher-order effect signature with no fork (i.e., no computation parameters):

$$\begin{aligned}
&\text{Lift} : \text{Effect} \to \text{Effect}^{\text{H}}\\
&\text{Op}^{\text{H}}\ \ (\text{Lift}\ \Delta) = \text{Op}\ \Delta\\
&\text{Ret}^{\text{H}}\ (\text{Lift}\ \Delta) = \text{Ret}\ \Delta\\
&\text{Fork}\ \ (\text{Lift}\ \Delta) = \lambda\ \_ \to \bot\\
&\text{Ty}\ \ \ \ (\text{Lift}\ \Delta) = \lambda()
\end{aligned}$$

Using this effect signature, and using higher-order effect row insertion witnesses analogous to the ones we defined and used in Section 3.2.2, the following smart constructor lets us represent any algebraic operation as a Hefty computation:

$$\uparrow\_ : \{\, w : \text{Lift}\ \Delta \lesssim^{\text{H}} H\,\} \to (op : \text{Op}\ \Delta) \to \text{Hefty}\ H\ (\text{Ret}\ \Delta\ op)$$

Using this notion of lifting, Hefty trees can be used to program against interfaces of both higher-order and plain algebraic effects.

3.3.3 *Higher-Order Operations with Polymorphic Return Types*

Let us consider how to define Catch as a higher-order effect. Ideally, we would define an operation that is parameterized by a return type of the branches of a particular catch operation, as shown on the left, such that we can define the higher-order effect signature on the right:[52]

[52] *d is for dubious.*

```
data CatchOpᵈ : Set₁ where
  catchᵈ : Set → CatchOpᵈ


Catchᵈ : Effectᴴ
Opᴴ  Catchᵈ = CatchOpᵈ
Retᴴ Catchᵈ (catchᵈ A) = A
Fork Catchᵈ (catchᵈ A) = Bool
Ty   Catchᵈ {catchᵈ A} _ = A
```

The Fork field on the right says that the Catch operation has two sub-computations (since Bool has two constructors), and that each computation parameter has some return type A. However, the signature on the right above is not well defined!

The problem is that, because CatchOp^d has a constructor that quantifies over a type (Set), the CatchOp^d type lives in Set$_1$. Consequently it does not fit the definition of Effect$^{\mathsf{H}}$, whose operations live in Set. There are two potential solutions to this problem: (1) increase the universe level of Effect$^{\mathsf{H}}$ to allow Op$^{\mathsf{H}}$ to live in Set$_1$; or (2) use a *universe of types* [Martin-Löf, 1984]. Either solution is applicable here. However, for some operations (e.g., λ in Section 3.4.1) it is natural to model types as an interface that we are programming against. For this reason, using a type universe is a natural fit.

A universe of types is a (dependent) pair of a syntax of types (Ty : Set) and a semantic function ($[\![\_]\!]^{\mathsf{T}}$: Ty → Set) defining the meaning of the syntax by reflecting it into Agda's Set:

```
record Univ : Set₁ where
  field Type : Set
        ⟦_⟧ᵀ : Type → Set
```

Using type universes, we can parameterize the catch constructor on the left below by a *syntactic type* Ty of some universe u, and use the *meaning of this type* ($[\![\ t\ ]\!]^{\mathsf{T}}$) as the return type of the computation parameters in the effect signature on the right below:

```
data CatchOp { u : Univ } : Set where
  catch : Type → CatchOp

Catch : { u : Univ } → Effect^H
Op^H  Catch            = CatchOp
Ret^H Catch (catch t)  = ⟦ t ⟧^T
Fork  Catch (catch t)  = Bool
Ty    Catch {catch t} = λ _ → ⟦ t ⟧^T
```

While the universe of types encoding restricts the kind of type that catch can have as a return type, the effect signature is parametric in the universe. Thus the implementer of the Catch effect signature (or interface) is free to choose a sufficiently expressive universe of types.

3.3.4 *Hefty Algebras*

As shown in Section 3.2.5, the higher-order catch operation can be encoded as a non-modular elaboration:

```
catch m₁ m₂
  = (♯ ((given hThrow handle m₁) tt)) ≫= (maybe pure m₂)
```

We can make this elaboration modular by expressing it as an *algebra* over Hefty trees containing operations of the Catch signature. To this end, we will use the following notion of hefty algebra ($\mathsf{Alg}^{\mathsf{H}}$) and fold (or *catamorphism* [Meijer et al., 1991], $\mathsf{cata}^{\mathsf{H}}$) for Hefty:

```
record Alg^H (H : Effect^H) (F : Set → Set) : Set₁ where
  field alg : ⟦ H ⟧^H F A → F A


cata^H : (∀ {A} → A → F A) → Alg^H H F → Hefty H A → F A
cata^H g a (pure x) = g x
cata^H g a (impure (op , k , s))
  = alg a (op , ((cata^H g a ∘ k) , (cata^H g a ∘ s)))
```

Here $\mathsf{Alg}^{\mathsf{H}}$ defines how to transform an impure node of type Hefty H A into a value of type F A, assuming we have already folded the computation parameters and continuation into F values. A nice property of algebras is that they are closed under higher-order effect signature sums:

```
_⋎_ : Algᴴ H₁ F → Algᴴ H₂ F → Algᴴ (H₁ ∔ H₂) F
alg (A₁ ⋎ A₂) (inj₁ op , k , s) = alg A₁ (op , k , s)
alg (A₁ ⋎ A₂) (inj₂ op , k , s) = alg A₂ (op , k , s)
```

By defining elaborations as hefty algebras (below) we can compose them using _⋎_.

```
Elaboration : Effectᴴ → Effect → Set₁
Elaboration H Δ = Algᴴ H (Free Δ)
```

An Elaboration H Δ elaborates higher-order operations of signature H into algebraic operations of signature Δ. Given an elaboration, we can generically transform any hefty tree into more primitive algebraic effects and handlers:

```
elaborate : Elaboration H Δ → Hefty H A → Free Δ A
elaborate = cataᴴ pure
```

EXAMPLE. The elaboration below is analogous to the non-modular catch elaboration discussed in Section 3.2.5 and in the beginning of this subsection:

```
eCatch : { u : Univ } { w : Throw ≲ Δ } → Elaboration Catch Δ
alg (eCatch { w = w }) (catch t , k , s) =
  (♯ ((given hThrow handle s true) tt)) >>= maybe k (s false >>= k)
```

The elaboration is essentially the same as its non-modular counterpart, except that it now uses the universe of types encoding discussed in Section 3.3.3, and that it now transforms syntactic representations of higher-order operations instead. Using this elaboration, we can, for example, run the following example program involving the state effect from Figure 8, the throw effect from Section 3.2.1, and the catch effect defined here:

```
transact :  { wₛ : Lift State ≲ᴴ H } { wₜ : Lift Throw ≲ᴴ H }
            { w : Catch ≲ᴴ H }
```

```
         → Hefty H ℕ
transact = do
  ↑ put 1
  ‵catch (do ↑ (put 2); (↑ throw) ≫= ⊥-elim) (pure tt)
  ↑ get
```

The program first sets the state to 1; then to 2; and then throws an exception. The exception is caught, and control is immediately passed to the final operation in the program which gets the state. By also defining elaborations for Lift and Nil, we can elaborate and run the program:

```
eTransact :  { _ : Throw ≲ Δ } { _ : State ≲ Δ }
          → Elaboration
               (Catch ∔ Lift Throw ∔ Lift State ∔ Lift Nil) Δ
eTransact = eCatch ⋎ eLift ⋎ eLift ⋎ eNil

test-transact : un ( ( given hSt
                       handle ( ( given hThrow
                                  handle (
                                    elaborate eTransact transact)
                                  ) tt ) )
                      0 ) ≡ (just 2 , 2)
test-transact = refl
```

The program above uses a so-called *global* interpretation of state, where the put operation in the “try block” of ‵catch causes the state to be updated globally. In Section 3.4.2.2 we return to this example and show how we can modularly change the elaboration of the higher-order effect Catch to yield a so-called *transactional* interpretation of state where the put operation in the try block is rolled back when an exception is thrown.

3.3.5 *Discussion and Limitations*

Which (higher-order) effects can we describe using hefty trees and algebras? Since the core mechanism of our approach is modular elaboration of higher-order operations into more primitive effects and handlers, it is clear that hefty trees and algebras are at least as expressive as standard algebraic effects. The crucial benefit of hefty algebras over algebraic effects is that higher-order operations can be declared and implemented modularly. In this sense, hefty algebras provide a modular abstraction layer over standard algebraic effects that, although it adds an extra layer of indirection by requiring both elaborations and handlers to give a semantics to hefty trees, is comparatively cheap and implemented using only standard techniques such as F-algebras. As we show in Section 3.5, hefty algebras also let us define higher-order effect theories, akin to algebraic effect theories.

Conceptually, we expect that hefty trees can capture any *monadic* higher-order effect whose signature is given by a higher-order functor on Set → Set. Filinski [1999] showed that any monadic effect can be represented using continuations, and given that we can encode the continuation monad using algebraic effects [Schrijvers et al., 2019] in terms of the *sub/jump* operations from Section 3.4.2.2 by Thielecke [1997], Fiore and Staton [2014], it is possible to elaborate any monadic effect into algebraic effects using hefty algebras. The current Agda implementation, though, is slightly more restrictive. The type of effect signatures, Effect^{H}, approximates the set of higher-order functors by constructively enforcing that all occurrences of the computation type are strictly positive. Hence, there may be higher-order effects that are well-defined semantically, but which cannot be captured in the Agda encoding presented here.

Recent work by Van den Berg and Schrijvers [2023] introduced a higher-order free monad that coincides with our Hefty type. Their work shows that hefty trees are, in fact, a free monad. Furthermore, they demonstrate that a range of existing effects frameworks from the literature can be viewed as instances of hefty trees.

When comparing hefty trees to scoped effects, we observe two important differences. The first difference is that the syntax of programs with higher-order effects is fundamentally more restrictive when using scoped effects. Specifically, as discussed at the end of Section 3.2.6.4, scoped effects impose a restriction on operations that their computation parameters must pass control directly to the continuation of the operation. Hefty trees, on the other hand, do not restrict the control flow of computation parameters, meaning that they can be used to define a broader class of operations. For instance, in Section 3.4.1 we define a higher-order effect for function abstraction, which is an example of an operation where control does not flow from the computation parameter to the continuation.

The second difference is that hefty algebras and scoped effects and handlers are modular in different ways. Scoped effects are modular because we can modularly define, compose, and handle scoped operations, by applying scoped effect handlers in sequence; i.e.:

$$\mathsf{Prog}\ \Delta_0\ \gamma_0\ A_0 \xrightarrow{h_1} \mathsf{Prog}\ \Delta_1\ \gamma_1\ A_1 \xrightarrow{h_2} \cdots \xrightarrow{h_n} \mathsf{Prog}\ \mathsf{Nil}\ \mathsf{Nil}\ A_n$$

As discussed in Section 3.2.6.3, each handler application modularly "weaves" effects through sub computations, using a dedicated glue function. Hefty algebras, on the other hand, work by applying an elaboration algebra assembled from modular components in one go. The program resulting from elaboration can then be handled using standard algebraic effect handlers; i.e.:

$$\begin{array}{l}
\mathsf{Hefty}\ (H_0 \mathbin{\dot{+}} \cdots \mathbin{\dot{+}} H_m)\ A \\
\quad \xrightarrow{\mathsf{elaborate}\ (E_0\ \curlyvee\ \cdots\ \curlyvee\ E_m)} \\
\mathsf{Free}\ \Delta\ A \\
\quad \xrightarrow{h_1} \cdots \xrightarrow{h_k} \\
\mathsf{Free}\ \mathsf{Nil}\ A_k
\end{array}$$

Because hefty algebras eagerly elaborate all higher-order effects in one go, they do not require similar "weaving" as scoped effect handlers. A consequence of this difference is that scoped effect handlers exhibit more effect interaction by default; i.e., different permutations of handlers may give different semantics. In contrast, when using hefty algebras we have to be more explicit about such effect interactions. We discuss this difference in more detail in Section 3.4.2.2.

3.4 EXAMPLES

As discussed in Section 3.2.5, there is a wide range of examples of higher-order effects that cannot be defined as algebraic operations directly, and are typically defined as non-modular elaborations instead. In this section we give examples of such effects and show to define them modularly using hefty algebras. The artifact [Van der Rest and Bach Poulsen, 2024] contains the full examples.

3.4.1 λ *as a Higher-Order Operation*

As recently observed by Van den Berg et al. [2021a], the (higher-order) operations for λ abstraction and application are neither algebraic nor scoped effects. We demonstrate how hefty algebras allow us to modularly define and elaborate an interface of higher-order operations for λ abstraction and application, inspired by Levy's call-by-push-value [Levy, 2004]. The interface we will consider is parametric in a universe of types given by the following record:

```
record LamUniv : Set₁ where
  field { u }   : Univ
        _↣_ : Type → Type → Type
        c       : Type → Type
```

The _↣_ field represents a function type, whereas c is the type of *thunk values*. Distinguishing thunks in this way allows us to assign either a call-by-value or call-by-name semantics to the interface for λ abstraction summarized by the following smart constructors:

```
‵lam :   {t₁ t₂ : Type} → (⟦ c t₁ ⟧ᵀ → Hefty H ⟦ t₂ ⟧ᵀ)
       → Hefty H ⟦ (c t₁) ↣ t₂ ⟧ᵀ
‵var  :   {t : Type}        → ⟦ c t ⟧ᵀ
       → Hefty H ⟦ t ⟧ᵀ
‵app :   {t₁ t₂ : Type} → ⟦ (c t₁) ↣ t₂ ⟧ᵀ → Hefty H ⟦ t₁ ⟧ᵀ
       → Hefty H ⟦ t₂ ⟧ᵀ
```

Here ‵lam is a higher-order operation with a function typed computation parameter and whose return type is a function value

($⟦$ c $t_1 \rightarrowtail t_2$ $⟧^{\mathsf{T}}$). The `var operation accepts a thunk value as argument and yields a value of a matching type. – The `app operation is also a higher-order operation: its first parameter is a function value type, whereas its second parameter is a computation parameter whose return type matches the function value parameter type.

The interface above defines a kind of *higher-order abstract syntax* [Pfenning and Elliott, 1988] which piggy-backs on Agda functions for name binding. However, unlike most Agda functions, the constructors above represent functions with side effects. The representation in principle supports functions with arbitrary side effects since it is parametric in what the higher-order effect signature H is. Furthermore, we can assign different operational interpretations to the operations in the interface without having to change the interface or programs written against the interface. To illustrate we give two different implementations of the interface: one that implements a call-by-value evaluation strategy, and one that implements call-by-name.

3.4.1.1 *Call-by-Value*

We give a call-by-value interpretation `lam by generically elaborating to algebraic effect trees with any set of effects Δ. Our interpretation is parametric in proof witnesses that the following isomorphisms hold for value types (↔ is the type of isomorphisms from the Agda standard library):[53]

[53] *The two sides of an isomorphism $A \leftrightarrow B$ are given by the functions* to *: $A \to B$ and* from *: $B \to A$.*

```
iso₁ : {t₁ t₂ : Type} → ⟦ t₁ ↣ t₂ ⟧ᵀ ↔ (⟦ t₁ ⟧ᵀ → Free Δ ⟦ t₂ ⟧ᵀ)
iso₂ : {t : Type}     → ⟦ c t     ⟧ᵀ ↔ ⟦ t ⟧ᵀ
```

The first isomorphism says that a function value type corresponds to a function which accepts a value of type t_1 and produces a computation whose return type matches the function type. The second says that thunk types coincide with value types. Using these isomorphisms, the following defines a call-by-value elaboration of functions:

```
eLamCBV : Elaboration Lam Δ
alg eLamCBV (lam , k , ψ) = k (from ψ)
alg eLamCBV (var $x$ , $k$ , _) = $k$ (to $x$)
alg eLamCBV (app $f$ , $k$ , $\psi$) = do
  $a$ ← $\psi$ tt
  $v$ ← to $f$ (from $a$)
  $k$ $v$
```

The lam case passes the function body given by the sub-tree ψ as a value to the continuation, where the from function mediates the sub-tree of type $[\![\, \mathsf{c}\ t_1 \,]\!]^{\mathsf{T}} \to \mathsf{Free}\ \Delta\ [\![\, t_2 \,]\!]^{\mathsf{T}}$ to a value type $[\![\, (\mathsf{c}\ t_1) \rightarrowtail t_2 \,]\!]^{\mathsf{T}}$, using the isomorphism iso_1. The var case uses the to function to mediate a $[\![\, \mathsf{c}\ t \,]\!]^{\mathsf{T}}$ value to a $[\![\, t \,]\!]^{\mathsf{T}}$ value, using the isomorphism iso_2. The app case first eagerly evaluates the argument expression of the application (in the sub-tree ψ) to an argument value, and then passes the resulting value to the function value of the application. The resulting value is passed to the continuation.

Using the elaboration above, we can evaluate programs such as the following which uses both the higher-order lambda effect, the algebraic state effect, and assumes that our universe has a number type $[\![\, num \,]\!]^{\mathsf{T}} \leftrightarrow \mathbb{N}$:

```
ex : Hefty (Lam ⊕ Lift State ⊕ Lift Nil) ℕ
ex = do
  ↑ put 1
  $f$ ← ‵lam (λ $x$ → do
         $n_1$ ← ‵var $x$
         $n_2$ ← ‵var $x$
         pure (from ((to $n_1$) + (to $n_2$))))
  $v$ ← ‵app $f$ incr
  pure (to $v$)
  where incr = do $s_0$ ← ↑ get
                  ↑ put ($s_0$ + 1)
                  $s_1$ ← ↑ get
                  pure (from $s_1$)
```

The program first sets the state to 1. Then it constructs a function that binds a variable x, dereferences the variable twice, and adds the two resulting values together. Finally, the application in the second-to-last line applies the function with an argument expression which increments the state by 1 and returns the resulting value. Running the program produces 4 since the state increment expression is eagerly evaluated before the function is applied.

```
elab-cbv : Elaboration (Lam ∔ Lift State ∔ Lift Nil) (State ⊕ Nil)
elab-cbv = eLamCBV ⋎ eLift ⋎ eNil

test-ex-cbv : un ((given hSt handle (elaborate elab-cbv ex)) 0)
                ≡ (4 , 2)
test-ex-cbv = refl
```

3.4.1.2 *Call-by-Name*

The key difference between the call-by-value and the call-by-name interpretation of our λ operations is that we now assume that thunks are computations. That is, we assume that the following isomorphisms hold for value types:

```
iso₁ : {t₁ t₂ : Type} → ⟦ t₁ ↣ t₂ ⟧ᵀ ↔ (⟦ t₁ ⟧ᵀ → Free Δ ⟦ t₂ ⟧ᵀ)
iso₂ : {t : Type}     → ⟦ c t ⟧ᵀ       ↔ Free Δ ⟦ t ⟧ᵀ
```

Using these isomorphisms, the following defines a call-by-name elaboration of functions:

```
eLamCBN : Elaboration Lam Δ
alg eLamCBN (lam , k , ψ) = k (from ψ)
alg eLamCBN (var x , k , _) = to x ≫= k
alg eLamCBN (app f , k , ψ) = to f (from (ψ tt)) ≫= k
```

The case for lam is the same as the call-by-value elaboration. The case for var now needs to force the thunk by running the computation and passing its result to k. The case for app passes the argument sub-tree (ψ) as an argument to the function f, runs the computation resulting from doing so, and then passes its result to k. Running the example program ex from above now produces 5 as result, since the state increment expression in the argument of ‘app is thunked and run twice during the evaluation of the called function.

```
elab-cbn : Elaboration (Lam ∔ Lift State ∔ Lift Nil) (State ⊕ Nil)
elab-cbn = eLamCBN ⋎ eLift ⋎ eNil

test-ex-cbn : un ((given hSt handle (elaborate elab-cbn ex)) 0)
                ≡ (5 , 3)
test-ex-cbn = refl
```

3.4.2 *Optionally Transactional Exception Catching*

A feature of scoped effect handlers [Wu et al., 2014, Piróg et al., 2018, Yang et al., 2022] is that changing the order of handlers makes it possible to obtain different semantics of *effect interaction*. A classical example of effect interaction is the interaction between state and exception catching that we briefly considered at the end of Section 3.3.4 in connection with this transact program:

```
transact : { w_s : Lift State ≲ᴴ H } { w_t : Lift Throw ≲ᴴ H }
           { w : Catch ≲ᴴ H }
         → Hefty H ℕ
transact = do
  ↑ put 1
  ‘catch (do ↑ put 2; (↑ throw) ≫= ⊥-elim) (pure tt)
  ↑ get
```

The state and exception catching effect can interact to give either of these two semantics:

1. *Global* interpretation of state, where the transact program returns 2 since the put operation in the "try" block causes the state to be updated globally.

2. *Transactional* interpretation of state, where the transact program returns 1 since the state changes of the put operation are *rolled back* when the "try" block throws an exception.

With monad transformers [Cenciarelli and Moggi, 1993, Liang et al., 1995b] we can recover either of these semantics by permuting the order of monad transformers. With scoped effect handlers we can also recover either by permuting the order of handlers.

$$
\begin{aligned}
\mathsf{h}\ (\text{`}\mathsf{sub}\ (\lambda\ \_ \to p)\ k) &\equiv \mathsf{h}\ p \\
\mathsf{h}\ (\text{`}\mathsf{sub}\ (\lambda\ r \to m \ggg\!= \text{`}\mathsf{jump}\ r)\ k) &\equiv \mathsf{h}\ (m \ggg\!= k) \\
\mathsf{h}\ (\text{`}\mathsf{sub}\ p\ (\text{`}\mathsf{jump}\ r')) &\equiv \mathsf{h}\ (p\ r') \\
\mathsf{h}\ (\text{`}\mathsf{sub}\ p\ q \ggg\!= k) &\equiv \mathsf{h}\ (\text{`}\mathsf{sub}\ (\lambda\ x \to p\ x \ggg\!= k\ )\ (\lambda\ x \to q\ x \ggg\!= k))
\end{aligned}
$$

Figure 10: Laws for the `sub and `jump operations

However, the eCatch elaboration in Section 3.3.4 always gives us a global interpretation of state. In this section we demonstrate how we can recover a transactional interpretation of state by using a different elaboration of the catch operation into an algebraically effectful program with the throw operation and the off-the-shelf *sub/jump* control effects due to Thielecke [1997], Fiore and Staton [2014]. The different elaboration is modular in the sense that we do not have to change the interface of the catch operation nor any programs written against the interface.

3.4.2.1 *Sub/Jump*

We recall how to define two operations, sub and jump, due to Thielecke [1997], Fiore and Staton [2014]. We define these operations as algebraic effects following Schrijvers et al. [2019]. The algebraic effects are summarized by the following smart constructors where $\mathsf{CC}\ \mathit{Ref}$ is associated with the sub/jump operations:

$$
\begin{aligned}
\text{`}\mathsf{sub}\ \ :\ & \{\, w : \mathsf{CC}\ \mathit{Ref} \lesssim \Delta \,\}\ (b : \mathit{Ref}\ t \to \mathsf{Free}\ \Delta\ A) \\
& \to (k : [\![\ t\ ]\!]^{\mathsf{T}} \to \mathsf{Free}\ \Delta\ A) \to \mathsf{Free}\ \Delta\ A
\end{aligned}
$$

$$
\text{`}\mathsf{jump} : \{\, w : \mathsf{CC}\ \mathit{Ref} \lesssim \Delta \,\}\ (\mathit{ref} : \mathit{Ref}\ t)\ (x : [\![\ t\ ]\!]^{\mathsf{T}}) \to \mathsf{Free}\ \Delta\ B
$$

An operation `sub $f$ $g$ gives a computation $f$ access to the continuation $g$ via a reference value $\mathit{Ref}\ t$ which represents a continuation expecting a value of type $[\![\ t\ ]\!]^{\mathsf{T}}$. The `jump operation invokes such continuations. The operations and their handler (abbreviated to h) satisfy the laws shown in Figure 10. The last law in Figure 10 asserts that `sub and `jump are *algebraic* operations, since their computational sub-terms behave as continuations. Thus, we encode `sub and its handler as an algebraic effect.

3.4.2.2 *Optionally Transactional Exception Catching*

By using the `sub and `jump operations in our elaboration of catch, we get a semantics of exception catching whose interaction with state depends on the order that the state effect and sub/jump effect is handled.

```
eCatchOT :   { w₁ : CC Ref ≲ Δ } { w₂ : Throw ≲ Δ }
           → Elaboration Catch Δ
alg eCatchOT (catch x , k , ψ) =
  let m₁ = ψ true; m₂ = ψ false in
  `sub (λ r → (♯ ((given hThrow handle m₁ ) tt))
                ≫= maybe k (`jump r (from tt)))
       (λ _ → m₂ ≫= k)
```

The elaboration uses `sub to capture the continuation of a higher-order catch operation. If no exception is raised, then control flows to the continuation $k$ without invoking the continuation of `sub. Otherwise, we jump to the continuation of `sub, which runs $m_2$ before passing control to $k$. Capturing the continuation in this way interacts with state because the continuation of `sub may have been pre-applied to a state handler that only knows about the "old" state. This happens when we invoke the state handler before the handler for sub/jump: in this case we get the transactional interpretation of state, so running transact gives 1. Otherwise, if we run the sub/jump handler before the state handler, we get the global interpretation of state and the result 2.

The sub/jump elaboration above is more involved than the scoped effect handler that we considered in Section 3.2.6. However, the complicated encoding does not pollute the higher-order effect interface, and only turns up if we strictly want or need effect interaction.

3.4.3 *Logic Programming*

Following Schrijvers et al. [2014], Wu et al. [2014], Yang et al. [2022] we can define a non-deterministic choice operation (_`or_) as an algebraic effect, to provide support for expressing the kind of non-deterministic search for solutions that is common in logic programming. We can also define a `fail operation which indicates that the search in the current branch was unsuccessful. The smart constructors below are the lifted higher-order counterparts to the `or and `fail operations:

```
_`orᴴ_ :  { Lift Choice ≲ᴴ H } → Hefty H A → Hefty H A
          → Hefty H A
`failᴴ   : { Lift Choice ≲ᴴ H } → Hefty H A
```

A useful operator for cutting non-deterministic search short when a solution is found is the `once operator. The `once operator is not an algebraic effect, but a scoped (and thus higher-order) effect.

```
`once :  { w : Once ≲ᴴ H } {t : Type}
       → Hefty H ⟦ t ⟧ᵀ → Hefty H ⟦ t ⟧ᵀ
```

We can define the meaning of the once operator as the following elaboration:

```
eOnce : { Choice ≲ Δ } → Elaboration Once Δ
alg eOnce (once , k , ψ) = do
  l ← ♯ ((given hChoice handle (ψ tt)) tt)
  maybe k `fail (head l)
```

The elaboration runs the branch (ψ) of once under the hChoice handler to compute a list of all solutions of ψ. It then tries to choose the first solution and pass that to the continuation *k*. If the branch has no solutions, we fail. Under a strict evaluation order, the elaboration computes all possible solutions which is doing more work than needed. Agda 2.6.2.2 does not have a specified evaluation strategy, but does compile to Haskell which is lazy. In Haskell, the solutions would be lazily computed, such that the once operator cuts search short as intended.

3.4.4 *Concurrency*

Finally, we consider how to define higher-order operations for concurrency, inspired by Yang et al.'s [2022] *resumption monad* (see also [Claessen, 1999, Schmidt, 1986, Piróg and Gibbons, 2014]) definition using scoped effects. We summarize our encoding and compare it with the resumption monad. The goal is to define the following operations:

```
`spawn  : {t : Type} → (m₁ m₂ : Hefty H ⟦ t ⟧ᵀ) → Hefty H ⟦ t ⟧ᵀ
`atomic : {t : Type} → Hefty H ⟦ t ⟧ᵀ           → Hefty H ⟦ t ⟧ᵀ
```

The operation `spawn $m_1$ $m_2$ spawns two threads that run concurrently, and returns the value produced by $m_1$ after both have finished. The operation `atomic m represents a block to be executed atomically; i.e., no other threads run before the block finishes executing.

We elaborate `spawn by interleaving the sub-trees of its computations. To this end, we use a dedicated function which interleaves the operations in two trees and yields as output the value of the left input tree (the first computation parameter):

```
interleaveₗ :  {Ref : Type → Set} → Free (CC Ref ⊕ Δ) A
            → Free (CC Ref ⊕ Δ) B → Free (CC Ref ⊕ Δ) A
```

Here, the CC effect is the sub/jump effect that we also used in Section 3.4.2.2. The interleave$_l$ function ensures atomic execution by only interleaving code that is not wrapped in a `sub operation. We elaborate Concur into CC as follows, where the to-front and from-front functions use the row insertion witness w_α to move the CC effect to the front of the row and back again:

```
eConcur : { w : CC Ref ≲ Δ } → Elaboration Concur Δ
alg eConcur (spawn t , k , ψ) =
  from-front (interleaveₗ (to-front (ψ true)) (to-front (ψ false)))
    ≫= k
alg eConcur (atomic t , k , ψ) =
  `sub (λ ref → ψ tt ≫= `jump ref) k
```

The elaboration uses `sub as a delimiter for blocks that should not be interleaved, such that the interleave$_l$ function only interleaves code that does not reside in atomic blocks. At the end of an atomic block, we `jump to the (possibly interleaved) computation context, *k*. By using `sub to explicitly delimit blocks that should not be interleaved, we have encoded what Wu et al. [2014, § 7] call *scoped syntax*.

EXAMPLE. Below is an example program that spawns two threads that use the Output effect. The first thread prints 0, 1, and 2; the second prints 3 and 4.

```
ex-01234 : Hefty (Lift Output ⊹ Concur ⊹ Lift Nil) ℕ
ex-01234 = `spawn (do ↑ out "0"; ↑ out "1"; ↑ out "2"; pure 0)
                  (do ↑ out "3"; ↑ out "4"; pure 0)
```

Since the Concur effect is elaborated to interleave the effects of the two threads, the printed output appears in interleaved order:

```
test-ex-01234 : un ( ( given hOut
                       handle
                         ( ( given hCC
                             handle
                               ( elaborate
                                   concur-elab ex-01234 )
                           ) tt ) ) tt ) ≡ (0 , "03142")
test-ex-01234 = refl
```

The following program spawns an additional thread with an `atomic block

```
ex-01234567 : Hefty (Lift Output ⊹ Concur ⊹ Lift Nil) ℕ
ex-01234567 =
  `spawn ex-01234
    (`atomic (do ↑ out "5"; ↑ out "6"; ↑ out "7"; pure 0))
```

Inspecting the output, we see that the additional thread indeed computes atomically:

```
test-ex-01234567 : un ( ( given hOut
                          handle
                            ( ( given hCC
                                  handle
                                    ( elaborate
                                        concur-elab ex-01234567 )
                                  ) tt ) ) tt ) ≡ (0 , "05673142")
test-ex-01234567 = refl
```

The example above is inspired by the resumption monad, and in particular by the scoped effects definition of concurrency due to Yang et al. [2022]. Yang et al. do not (explicitly) consider how to define the concurrency operations in a modular style. Instead, they give a direct semantics that translates to the resumption monad which we can encode as follows in Agda (assuming resumptions are given by the free monad):

```
data Resumption Δ A : Set where
  done : A                          → Resumption Δ A
  more : Free Δ (Resumption Δ A) → Resumption Δ A
```

We could elaborate into this type using a hefty algebra $\mathsf{Alg}^{\mathsf{H}}$ Concur (Resumption Δ) but that would be incompatible with our other elaborations which use the free monad. For that reason, we emulate the resumption monad using the free monad instead of using the Resumption type directly.

3.5 MODULAR REASONING FOR HIGHER-ORDER EFFECTS

A key aspect of algebraic effects and handlers is the ability to state and prove *equational laws* that characterize correct implementations of effectful operations. Usually, an effect comes equipped with multiple laws that govern its intended behavior. An effect and its laws is generally known as as *effect theory* [Hyland et al., 2006, Plotkin and Power, 2002, 2003, Yang and Wu, 2021]. This concept of effect theory extends to *higher-order effect theories*, which describe the intended behavior of higher-order effects. In this section, we first discuss how to define theories for algebraic effects in Agda by adapting the exposition of Yang and Wu [2021], and show how correctness of implementations with respect to a given theory can be stated and proved. We then extend this reasoning infrastructure to higher-order effects, allowing for modular reasoning about the correctness of elaborations of higher-order effects.

Let us consider the state effect as an example, which comprises the get and put operations. With the state effect, we typically associate a set of equations (or laws) that specify how its implementations ought to behave. One such law is the *get-get* law, which captures the intuition that the state returned by two subsequent get operation does not change if we do not use the put operation in between:

$$\text{`get} \gg\!= \lambda s \to \text{`get} \gg\!= \lambda s' \to k\ s\ s' \;\equiv\; \text{`get} \gg\!= \lambda s \to k\ s\ s$$

We an define equational laws for higher-order effects in a similar fashion. For example, the following *catch-return* law for the `catch operation of the Catch effect, stating that catching exceptions in a computation that only returns a value does nothing.

$$\text{`catch}\ (\text{pure}\ x)\ m \;\equiv\; \text{pure}\ x$$

Correctness of an implementation of an algebraic effect with respect to a given theory is defined by comparing the implementations of programs that are equal under that theory. That is, if we can show that two programs are equal using the equations of a theory for its effects, handling the effects should produce equal results. For instance, a way to implement the state effect is by mapping programs to functions of the form $S \to S \times A$. Such an implementation would be correct if programs that are equal with respect to a theory of the state effect are mapped to functions that give the same value and output state for every input state.

For higher-order effects, correctness is defined in a similar manner. However, since we define higher-order effects by elaborating them into algebraic effects, correctness of elaborations with respect to a higher-order effect theory is defined by comparing the elaborated programs. Crucially, the elaborated programs do not have to be syntactically equal, but rather we should be able to prove them equal using a theory of the algebraic effects used to implement a higher-order effect.

Effect theories are known to be closed under the co-product of effects, by combining the equations into a new theory that contains all equations for both effects [Hyland et al., 2006]. Similarly, theories of higher-order effects are closed under sums of higher-order effect signatures. In Section 3.5.8, we show that composing two elaborations preserves their correctness, with respect to the sum of their respective theories.

3.5.1 *Theories of Algebraic Effects*

Theories of effects are collections of equations, so we start defining the type of equations in Agda. At its core, an equation for an effect Δ is given by a pair of effect trees of type Free Δ A, that define the left- and right-hand side of the equation. However, looking at the *get-get* law above, we see that this equation contains a *term metavariable*; i.e., k. Furthermore, when considering the type of k, which is $S \to S \to$ Free Δ A, we see that it refers to a *type metavariable*; i.e., A. Generally speaking, an equation may refer to any number of term metavariables, which, in turn, may depend on any number of type metavariables. Moreover, the type of the value returned by the left hand side and right hand side of an equation may depend on these type metavariables as well, as is the case for the *get-get* law above. This motivates the following definition of equations in Agda.

```
record Equation (Δ : Effect) : Set₁ where
  field
    V       : ℕ
    Γ       : Vec Set V → Set
    R       : Vec Set V → Set
    lhs rhs : (vs : Vec Set V) → Γ vs → Free Δ (R vs)
```

An equation consists of five components. The field V defines the number of type metavariables used in the equation. Then, the fields Γ and R define the term metavariables respectively return type of the equation. Both may depend on the type metavariables of the equation, hence they depend on a vector of length V containing unapplied substitutions for all type metavariables.

Finally, the left-hand side (lhs) and right-hand side (rhs) of the equation are then defined as functions of type Free Δ (R vs), which depend on unapplied substitutions for both the type and term level metavariables that the equation can refer to.

EXAMPLE . To illustrate how the Equation record captures equational laws of effects, we consider how to define the *get-get* as a value of type Equation State. Recall that the equation depends on one type metavariable, and one term metavariable. Furthermore, the return type of the programs on the left and right hand sides should be equal to this type metavariable.

```
get-get : Equation State
V  get-get = 1
Γ  get-get = λ where (A :: []) → ℕ → ℕ → Free State A
R  get-get = λ where (A :: []) → A
```

Since there is exactly one term metavariable, we equate Γ to the type of this metavariable. For equations with more than one metavariable, we would define Γ as a product of the types of all term metavariables. The fields lhs and rhs for the *get-get* law are then defined as follows:

```
lhs get-get (A :: []) k = `get >>= λ s → `get >>= λ s′ → k s s′
rhs get-get (A :: []) k = `get >>= λ s → k s s
```

3.5.2 *Modal Necessity*

The current definition of equations is too weak, in the sense that it does not apply in many situations where it should. The issue is that it fixes the set of effects that can be used in the left- and right-hand side. To illustrate why this is problematic, consider the following equality:

$$\text{get} \gg\!= \lambda s \to \text{get} \gg\!= \lambda s' \to \text{throw} \;\equiv\; \text{get} \gg\!= \lambda s \to \text{throw} \qquad (1)$$

We might expect to be able to prove this equality using the *get-get* law, but using the embedding of the law defined above—i.e., get-get—this is not possible. The reason for this is that we cannot pick an appropriate instantiation for the term metavariable k: it ranges over values of type $S \to S \to$ Free State A, inhibiting all references to effectful operation that are not part of the state effect, such as throw.

Given an equation for the effect Δ, the solution to this problem is to view Δ as a *lower bound* on the effects that might occur in the left-hand and right-hand side of the equation, rather than an exact specification. Effectively, this means that we close over all posible contexts of effects in which the equation can occur. This pattern of closing over all possible extensions of a type index is well-known [Allais et al., 2021, Van der Rest et al., 2022b], and corresponds to a shallow embedding of the Kripke semantics of the necessity modality from modal logic. We can define it in Agda as follows.

```
record □ (P : Effect → Set₁) (Δ : Effect) : Set₁ where
  constructor necessary
  field
    □⟨_⟩ : ∀ {Δ′} → { Δ ≲ Δ′ } → P Δ′
```

Intuitively, the □ modality transforms, for any effect-indexed type (P : Effect → Set$_1$), an *exact* specification of the set of effects to a *lower bound* on the set of effects. For equations, the difference between terms of type Equation Δ and □ Equation Δ amounts to the former defining an equation relating programs that have exactly effects Δ, while the latter defines an equation relating programs that have at least the effects Δ but potentially more. The □ modality is a *comonad*: the counit (extract below) witnesses that we can always transform a lower bound on effects to an exact specification, by instantiating the extension witness with a proof of reflexivity.

```
extract : {P : Effect → Set₁} → □ P Δ → P Δ
extract px = □⟨ px ⟩ { ≲-refl }
```

We can now redefine the *get-get* law such that it applies to all programs that have the State effect, but potentially other effects too.

```
get-get : □ Equation State
V   □⟨ get-get ⟩ = 1
Γ   □⟨ get-get ⟩ (A :: [])   = ℕ → ℕ → Free _ A
R   □⟨ get-get ⟩ (A :: [])   = A
lhs □⟨ get-get ⟩ (A :: []) k = `get ≫= λ s → `get ≫= λ s′ → k s s′
rhs □⟨ get-get ⟩ (A :: []) k = `get ≫= λ s → k s s
```

The above definition of the *get-get* law now lets us prove the equality in Equation (1); the term metavariable k ranges ranges over all continuations that return a tree of type Free Δ' A, for all Δ' such that State $\lesssim \Delta'$. This way, we can instantiate Δ' with an effect signature that subsumes both the State and the Throw, which in turn allows us to instantiate k with throw.

3.5.3 *Effect Theories*

Equations for an effect Δ can be combined into a *theory* for Δ. A theory for the effect Δ is simply a collection of equations, transformed using the □ modality to ensure that term metavariables can range over programs that include more effects than just Δ.

```
record Theory (Δ : Effect) : Set₁ where
  field
    arity     : Set
    equations : arity → □ Equation Δ
```

An effect theory consists of an arity, that defines the number of equations in the theory, and a function that maps arities to equations. We can think of effect theories as defining a specification for how implementations of an effect ought to behave. Although implementations may vary, depending for example on whether they are tailored to readability or efficiency, they should at least respect the equations of the theory of the effect they implement. We will make precise what it means for an implementation to respect an equation in Section 3.5.5.

Effect theories are closed under several composition operations that allow us to combine the equations of different theories into single theory. The most basic way of combining effect theories is by summing their arities.

$$
\begin{aligned}
&\_\langle+\rangle\_ : \mathsf{Theory}\ \Delta \to \mathsf{Theory}\ \Delta \to \mathsf{Theory}\ \Delta\\
&\mathsf{arity}\quad\ \ (T_1\ \langle+\rangle\ T_2) = \mathsf{arity}\ T_1 \uplus \mathsf{arity}\ T_2\\
&\mathsf{equations}\ (T_1\ \langle+\rangle\ T_2)\ (\mathsf{inj}_1\ a) = \mathsf{equations}\ T_1\ a\\
&\mathsf{equations}\ (T_1\ \langle+\rangle\ T_2)\ (\mathsf{inj}_2\ a) = \mathsf{equations}\ T_2\ a
\end{aligned}
$$

This way of combining effects is somewhat limiting, as it imposes that the theories we are combining are theories for the exact same effect. It is more likely, however, that we would want to combine theories for different effects. This requires that we can *weaken* effect theories with respect to the $\_\lesssim\_$ relation.

$$
\begin{aligned}
&\mathsf{weaken}\text{-}\Box :\quad \{P : \mathsf{Effect} \to \mathsf{Set}_1\} \to \{\ \Delta_1 \lesssim \Delta_2\ \}\\
&\qquad\qquad\ \to \Box\ P\ \Delta_1 \to \Box\ P\ \Delta_2\\
&\Box\langle\ \mathsf{weaken}\text{-}\Box\ \{\ w\ \}\ px\ \rangle\ \{\ w'\ \} = \Box\langle\ px\ \rangle\ \{\ \lesssim\text{-trans}\ w\ w'\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{weaken\text{-}theory} : \{\ \Delta_1 \lesssim \Delta_2\ \} \to \mathsf{Theory}\ \Delta_1 \to \mathsf{Theory}\ \Delta_2\\
&\mathsf{arity}\qquad\ (\mathsf{weaken\text{-}theory}\ T) = \mathsf{arity}\ T\\
&\mathsf{equations}\ (\mathsf{weaken\text{-}theory}\ T) = \lambda\ a \to \mathsf{weaken}\text{-}\Box\ (T\ .\mathsf{equations}\ a)
\end{aligned}
$$

Categorically speaking, the observation that for a given effect-indexed type P we can transform a value of type P Δ_1 to a value of type P Δ_2 if we know that $\Delta_1 \lesssim \Delta_2$ is equivalent to saying that P is a functor from the category of containers and container morphisms to the categorie of sets. From this perspective, the existence of weakening for free Free, as witnessed by the $\sharp$ operation discussed in Section 3.3 implies that it too is a such a functor.

With weakening for theories at our disposal, we can combine effect theories for different effects into a theory of the coproduct of their respective effects. This requires us to first define appropriate witnesses relating coproducts to effect inclusion.

$$
\begin{aligned}
&\lesssim\text{-}\oplus\text{-left}\ \ \ : \Delta_1 \lesssim (\Delta_1 \oplus \Delta_2)\\
&\lesssim\text{-}\oplus\text{-right} : \Delta_2 \lesssim (\Delta_1 \oplus \Delta_2)
\end{aligned}
$$

It is now straightforward to show that effect theories are closed under the coproduct of effect signatures, by summing the weakened theories.

$$
\begin{aligned}
&\_[+]\_ : \mathsf{Theory}\ \Delta_1 \to \mathsf{Theory}\ \Delta_2 \to \mathsf{Theory}\ (\Delta_1 \oplus \Delta_2)\\
&T_1\ [+]\ T_2 =
\end{aligned}
$$

```
        weaken-theory { ≲-⊕-left } T₁
    ⟨+⟩ weaken-theory { ≲-⊕-right } T₂
```

While this operation is in principle sufficient for our purposes, it forces a specific order on the effects of the combined theories. We can further generalize the operation above to allow for the effects of the combined theory to appear in any order. This requires the following instances.

```
≲-•-left  : { Δ₁ • Δ₂ ≈ Δ } → Δ₁ ≲ Δ
≲-•-right : { Δ₁ • Δ₂ ≈ Δ } → Δ₂ ≲ Δ
```

We show that effect theories are closed under coproducts up to reordering by, again, summing the weakened theories.

```
compose-theory :  { Δ₁ • Δ₂ ≈ Δ }
               → Theory Δ₁ → Theory Δ₂ → Theory Δ
compose-theory T₁ T₂ =
        weaken-theory { ≲-•-left } T₁
    ⟨+⟩ weaken-theory { ≲-•-right } T₂
```

Since equations are defined by storing the syntax trees that define their left-hand and right-hand side, and effect trees are weakenable, we would expect equations to be weakenable too. Indeed, we can define the following function witnessing weakenability of equations.

```
weaken-eq : { Δ₁ ≲ Δ₂ } → Equation Δ₁ → Equation Δ₂
```

This begs the question: why would we opt to use weakenability of the $\Box$ modality (or, bother with the $\Box$ modality at all) to show that theories are weakenable, rather than using weaken-eq directly? Although the latter approach would indeed allow us to define the composition operations for effect theories defined above, the possible ways in which we can instantiate term metavariables remains too restrictive. That is, we still would not be able to prove the equality in Equation (1), despite the fact that we can weaken the *get-get* law so that it applies to programs that use the Throw effect as well. Instantiations of the term metavariable k will be limited to weakened effect trees, precluding any instantiation that use operations of effects other than State, such as throw.

Finally, we define the following predicate to witness that an equation is part of a theory.

```
_◀_ : □ Equation Δ → Theory Δ → Set₁
eq ◀ T = ∃ λ a → T .equations a ≡ eq
```

We construct a proof *eq* ◀ *T* that an equation *eq* is part of a theory *T* by providing an arity together with a proof that *T* maps to *eq* for that arity.

3.5.4 *Syntactic Equivalence of Effectful Programs*

As discussed, propositional equality of effectful programs is too strict, as it precludes us from proving equalities that rely on a semantic understanding of the effects involved, such as the equality in Equation (1). The solution is to define an inductive relation that captures syntactic equivalence modulo some effect theory. We base our definition of syntactic equality of effectful programs on the relation defining equivalent computations by Yang and Wu [2021], Definition 3.1, adapting their definition where necessary to account for the use of modal necessity in the definition of Theory.

```
data _≈⟨_⟩_ {Δ Δ′} {_ : Δ ≲ Δ′ } :  (m₁ : Free Δ′ A)
                                  → Theory Δ
                                  → (m₂ : Free Δ′ A)
                                  → Set₁ where
```

A value of type $m_1 \approx\langle\, T \,\rangle\, m_2$ witnesses that programs m_1 and m_2 are equal modulo the equations of theory T. The first three constructors ensure that it is an equivalence relation.

```
≈-refl  : m ≈⟨ T ⟩ m
≈-sym   : m₁ ≈⟨ T ⟩ m₂ → m₂ ≈⟨ T ⟩ m₁
≈-trans : m₁ ≈⟨ T ⟩ m₂ → m₂ ≈⟨ T ⟩ m₃ → m₁ ≈⟨ T ⟩ m₃
```

Then, we add the following congruence rule, that establish that we can prove equality of two programs starting with the same operation by proving that the continuations yield equal programs for every possible value.

```
≈-cong :   (op : Op Δ′)
         → (k₁ k₂ : Ret Δ′ op → Free Δ′ A)
         → (∀ x → k₁ x ≈⟨ T ⟩ k₂ x)
         → impure (op , k₁) ≈⟨ T ⟩ impure (op , k₂)
```

The final constructor allows to prove equality of programs by reifying equations of an effect theory.

```
≈-eq :   (eq : □ Equation Δ)
       → (px : eq ◀ T)
       → (vs : Vec Set (V (□⟨ eq ⟩)))
       → (γ : Γ (□⟨ eq ⟩) vs)
       → (k : R (□⟨ eq ⟩) vs → Free Δ′ A)
       →          (lhs (□⟨ eq ⟩) vs γ ≫= k)
         ≈⟨ T ⟩ (rhs (□⟨ eq ⟩) vs γ ≫= k)
```

Since the equations of a theory are wrapped in the □ modality, we cannot refer to its components directly, but we must first provide a suitable extension witness.

With the ≈-eq constructor, we can prove equivalence between the left-hand and right-hand side of an equation, sequenced with an arbitrary continuation *k*. For convenience, we define the following lemma that allows us to apply an equation where the sides of the equation do not have a continuation.

```
use-equation :   { _ : Δ ≲ Δ′ }
               → {T : Theory Δ}
               → (eq : □ Equation Δ)
               → eq ◀ T
               → (vs : Vec Set (V □⟨ eq ⟩))
               → {γ : Γ (□⟨ eq ⟩) vs}
               → lhs (□⟨ eq ⟩) vs γ ≈⟨ T ⟩ rhs (□⟨ eq ⟩) vs γ
```

The definition of use-equation follows readily from the right-identity law for monads, i.e., $m \gg= \text{pure} \equiv m$, which allows us to instantiate ≈-eq with pure.

To construct proofs of equality it is convenient to use the following set of combinators to write proof terms in an equational style. They are completely analogous to the combinators commonly used to construct proofs of Agda's propositional equality, for example, as found in PLFA [Wadler et al., 2020].

```
module ≈-Reasoning (T : Theory Δ) { _ : Δ ≲ Δ′ } where

  begin_ :  {m₁ m₂ : Free Δ′ A}
         → m₁ ≈⟨ T ⟩ m₂ → m₁ ≈⟨ T ⟩ m₂
  begin eq = eq

  _■ : (m : Free Δ′ A) → m ≈⟨ T ⟩ m
  m ■ = ≈-refl

  _≈⟪⟫_ :  (m₁ : Free Δ′ A) {m₂ : Free Δ′ A}
         → m₁ ≈⟨ T ⟩ m₂ → m₁ ≈⟨ T ⟩ m₂
  m₁ ≈⟪⟫ eq = eq

  _≈⟪_⟫_ :  (m₁ {m₂ m₃} : Free Δ′ A)
          → m₁ ≈⟨ T ⟩ m₂ → m₂ ≈⟨ T ⟩ m₃ → m₁ ≈⟨ T ⟩ m₃
  m₁ ≈⟪ eq₁ ⟫ eq₂ = ≈-trans eq₁ eq₂
```

We now have all the necessary tools to prove syntactic equality of programs modulo a theory of their effect. To illustrate, we consider how to prove the equation in Equation (1). First, we define a theory for the State effect containing the get-get◀ law. While this is not the only law typically associated with State, for this example it is enough to only have the get-get law.

```
StateTheory : Theory State
arity StateTheory           = ⊤
equations StateTheory tt = get-get
```

Now to prove the equality in Equation (1) is simply a matter of invoking the get-get law.

```
get-get-throw :
     { _ : Throw ≲ Δ } { _ : State ≲ Δ }
  → (`get ≫= λ s → `get ≫= λ s′ → `throw {A = A})
     ≈⟨ StateTheory ⟩ (`get ≫= λ s → `throw)
get-get-throw {A = A} = begin
```

```
  `get >>= (λ s → `get >>= (λ s′ → `throw))
≈⟪ use-equation get-get (tt , refl) (A :: []) ⟫
  `get >>= (λ s → `throw)
■
where open ≈-Reasoning StateTheory
```

3.5.5 *Handler Correctness*

A handler is correct with respect to a given theory if handling syntactically equal programs yields equal results. Since handlers are defined as algebras over effect signatures, we start by defining what it means for an algebra of an effect Δ to respect an equation of the same effect, adapting Definition 2.1 from the exposition of Yang and Wu [2021].

```
Respects : Alg Δ A → Equation Δ → Set₁
Respects alg eq = ∀ {vs γ k} →
  fold k alg (lhs eq vs γ) ≡ fold k alg (rhs eq vs γ)
```

An algebra *alg* respects an equation *eq* if folding with that algebra produces propositionally equal results for the left- and right-hand side of the equation, for all possible instantiations of its type and term metavariables, and continuations k.

A handler *H* is correct with respect to a given theory *T* if its algebra respects all equations of *T* [Yang and Wu, 2021, Definition 4.3].

```
Correct : {P : Set} → Theory Δ → ⟨ A ! Δ ⇒ P ⇒ B ! Δ′ ⟩ → Set₁
Correct T H = ∀ {eq} → eq ◂ T → Respects (H .hdl) (extract eq)
```

We can now show that the handler for the State effect defined in Figure 8 is correct with respect to StateTheory. The proof follows immediately by reflexivity.

```
hStCorrect : Correct {A = A} {Δ′ = Δ} StateTheory hSt
hStCorrect (tt , refl) {_ :: []} {γ = k} = refl
```

3.5.6 *Theories of Higher-Order Effects*

For the most part, equations and theories for higher-order effects are defined in the same way as for first-order effects and support many of the same operations. Indeed, the definition of equations ranging over higher-order effects is exactly the same as its first-order counterpart, the most major difference being that the left-hand and right-hand side are now defined as Hefty trees. To ensure compatibility with the use of type universes to avoid size-issues, we must also allow type metavariables to range over the types in a universe in addition to Set. For this reason, the set of type metavariables is no longer described by a natural number, but rather by a list of kinds, which stores for each type metavariable whether it ranges over a types in a universe, or an Agda Set.

```
data Kind : Set where set type : Kind
```

A TypeContext carries unapplied substitutions for a given set of type metavariables, and is defined by induction over a list of kinds.

```
TypeContext : List Kind → Set₁
TypeContext []           = Level.Lift _ ⊤
TypeContext (set  :: vs) = Set × TypeContext vs
TypeContext (type :: vs) =
  Level.Lift (sℓ 0ℓ) Type × TypeContext vs
```

Equations of higher-order effects are then defined as follows.

```
record Equationᴴ (H : Effectᴴ) : Set₁ where
  field
    V       : List Kind
    Γ       : TypeContext V → Set
    R       : TypeContext V → Set
    lhs rhs : (vs : TypeContext V) → Γ vs → Hefty H (R vs)
```

This definition of equations suffers the same problem when it comes to term metavariables, which here too can only range over programs that exhibit the exact effect that the equation is defined for. Again, we address the issue using an embedding of modal necessity to close over all possible extensions of this effect. The definition is analogous to the one in Section 3.5.2, but this time we use higher-order effect subtyping as the modal accessibility relation:

```
record □ (P : Effectᴴ → Set₁) (H : Effectᴴ) : Set₁ where
  constructor necessary
  field □⟨_⟩ : ∀ {H′} → { H ≲ᴴ H′ } → P H′
```

To illustrate: we can define the *catch-return* law from the introduction of this section as a value of type □ Equationᴴ Catch a follows. Since the ‛catch operation relies on a type universe to avoid size issues, the sole type metavariable of this equation must range over the types in this universe as well.

```
catch-return : □ Equationᴴ Catch
V   □⟨ catch-return ⟩                  = type :: []
Γ   □⟨ catch-return ⟩ (lift t , _) = ⟦ t ⟧ᵀ × Hefty _ ⟦ t ⟧ᵀ
R   □⟨ catch-return ⟩ (lift t , _) = ⟦ t ⟧ᵀ
lhs □⟨ catch-return ⟩ _ (x , m)  = ‛catch (pure x) m
rhs □⟨ catch-return ⟩ _ (x , m)  = pure x
```

Theories of higher-order effects bundle extensible equations. The setup is the same as for theories of first-order effects.

```
record Theoryᴴ (H : Effectᴴ) : Set₁ where
  field
    arity : Set
    equations : arity → □ Equationᴴ H
```

The following predicate establishes that an equation is part of a theory. We prove this fact by providing an arity whose corresponding equation is equal to *eq*.

```
_◀ᴴ_ : □ Equationᴴ H → Theoryᴴ H → Set₁
eq ◀ᴴ Th = ∃ λ a → eq ≡ equations Th a
```

Weakenability of theories of higher-order effects then follows from weakenability of its equations.

```
weaken-□ : ∀ {P} → { H₁ ≲ᴴ H₂ } → □ P H₁ → □ P H₂
□⟨ weaken-□ { w } px ⟩ { w′ } = □⟨ px ⟩ { ≲ᴴ-trans w w′ }

weaken-theoryᴴ : { H₁ ≲ᴴ H₂ } → Theoryᴴ H₁ → Theoryᴴ H₂
arity     (weaken-theoryᴴ Th)   = Th .arity
equations (weaken-theoryᴴ Th) a = weaken-□ (Th .equations a)
```

Theories of higher-order effects can be combined using the following sum operation. The resulting theory contains all equations of both argument theories.

```
_⟨+⟩ᴴ_ : ∀[ Theoryᴴ ⇒ Theoryᴴ ⇒ Theoryᴴ ]
arity     (Th₁ ⟨+⟩ᴴ Th₂)          = arity Th₁ ⊎ arity Th₂
equations (Th₁ ⟨+⟩ᴴ Th₂) (inj₁ a) = equations Th₁ a
equations (Th₁ ⟨+⟩ᴴ Th₂) (inj₂ a) = equations Th₂ a
```

Theories of higher-order effects are closed under sums of higher-order effect theories as well. This operation is defined by appropriately weakening the respective theories, for which we need the following lemmas witnessing that higher-order effect signatures can be injected in a sum of signatures.

```
≲-∔-left  : H₁ ≲ᴴ (H₁ ∔ H₂)
≲-∔-right : H₂ ≲ᴴ (H₁ ∔ H₂)
```

The operation that combines theories under signature sums is then defined like so.

```
_[+]ᴴ_ : Theoryᴴ H₁ → Theoryᴴ H₂ → Theoryᴴ (H₁ ∔ H₂)
Th₁ [+]ᴴ Th₂ =
         weaken-theoryᴴ { ≲-∔-left } Th₁
  ⟨+⟩ᴴ weaken-theoryᴴ { ≲-∔-right } Th₂
```

3.5.7 *Equivalence of Programs with Higher-Order Effects*

We define the following inductive relation to capture equivalence of programs with higher-order effects modulo the equations of a given theory.

```
data _≅⟨_⟩_ {_ : H₁ ≲ᴴ H₂} : (m₁ : Hefty H₂ A)
                          → Theoryᴴ H₁
                          → (m₂ : Hefty H₂ A)
                          → Set₁ where
```

To ensure that it is indeed an equivalence relation, we include constructors for reflexivity, symmetry, and transitivity.

```
  ≅-refl  : ∀ {m : Hefty H₂ A}
          → m ≅⟨ Th ⟩ m

  ≅-sym   : ∀ {m₁ : Hefty H₂ A} {m₂}
          → m₁ ≅⟨ Th ⟩ m₂
          → m₂ ≅⟨ Th ⟩ m₁

  ≅-trans : ∀ {m₁ : Hefty H₂ A} {m₂ m₃}
          → m₁ ≅⟨ Th ⟩ m₂ → m₂ ≅⟨ Th ⟩ m₃
          → m₁ ≅⟨ Th ⟩ m₃
```

Furthermore, we include the following congruence rule that equates two program trees that have the same operation at the root, if their continuations are equivalent for all inputs.

```
  ≅-cong  :   (op : Opᴴ H₂)
          → (k₁ k₂ : Retᴴ H₂ op → Hefty H₂ A)
          → (s₁ s₂ : (ψ : Fork H₂ op) → Hefty H₂ (Ty H₂ ψ))
          → (∀ {x} → k₁ x ≅⟨ Th ⟩ k₂ x)
          → (∀ {ψ} → s₁ ψ ≅⟨ Th ⟩ s₂ ψ)
          → impure (op , k₁ , s₁) ≅⟨ Th ⟩ impure ( op , k₂ , s₂ )
```

Finally, we include a constructor that equates two programs using an equation of the theory.

```
  ≅-eq    :   (eq : □ Equationᴴ H₁)
          → eq ◂ᴴ Th
          → (vs : TypeContext (V □⟨ eq ⟩))
          → (γ : Γ □⟨ eq ⟩ vs)
          → (k : R □⟨ eq ⟩ vs → Hefty H₂ A)
```

```
→              (lhs □⟨ eq ⟩ vs γ ≫= k)
    ≅⟨ Th ⟩ (rhs □⟨ eq ⟩ vs γ ≫= k)
```

We can define the same reasoning combinators to construct proofs of equivalence for programs with higher-order effects.

```
module ≅-Reasoning { _ : H₁ ≲ᴴ H₂ } (Th : Theoryᴴ H₁) where

  begin_     :  {m₁ m₂ : Hefty H₂ A}
             → m₁ ≅⟨ Th ⟩ m₂ → m₁ ≅⟨ Th ⟩ m₂
  begin eq = eq

  _■ : (c : Hefty H₂ A) → c ≅⟨ Th ⟩ c
  c ■ = ≅-refl

  _≅⟨⟩_ :  (m₁ : Hefty H₂ A) {m₂ : Hefty H₂ A}
          → m₁ ≅⟨ Th ⟩ m₂ → m₁ ≅⟨ Th ⟩ m₂
  c₁ ≅⟨⟩ eq = eq

  _≅⟨_⟩_ :  (c₁ {c₂ c₃} : Hefty H₂ A)
             → c₁ ≅⟨ Th ⟩ c₂ → c₂ ≅⟨ Th ⟩ c₃ → c₁ ≅⟨ Th ⟩ c₃
  c₁ ≅⟨ eq₁ ⟩ eq₂ = ≅-trans eq₁ eq₂
```

To illustrate, we can prove that the programs catch throw (censor f m) and censor f m are equal under a theory for the afCatch effect that contains the *catch-return* law.

```
catch-return-censor : ∀ {t : Type} {f} {x : ⟦ t ⟧ᵀ}
                        {m : Hefty H ⟦ t ⟧ᵀ}
                      → { _ : Catch ≲ᴴ H }
                      → { _ : Censor ≲ᴴ H }
                      → ‵catch (pure x) (‵censor f m)
                        ≅⟨ CatchTheory ⟩ pure x
catch-return-censor {f = f} {x = x} {m = m} =
  begin
    ‵catch (pure x) (‵censor f m)
  ≅⟨ use-equationᴴ catch-return (tt , refl) _ ⟩
    pure x
  ■
  where open ≅-Reasoning _
```

The equivalence proof above makes, again, essential use of modal necessity. That is, by closing over all possible extensions of the Catch effe, the term metavariable in the *catch-return* law to range over programs that have higher-order effects other than Catch, which is needed to apply the law if the second branch of the catch operation contains the censor operation.

3.5.8 *Correctness of Elaborations*

As the first step towards defining correctness of elaborations, we must specify what it means for an algebra over a higher-order effect signature H to respect an equation. The definition is broadly similar to its counterpart for first-order effects in Section 3.5.5, with the crucial difference that the definition of "being equation respecting" for algebras over higher-order effect signatures is parameterized over a binary relation _≈_ between first-order effect trees. In practice, this binary relation will be instantiated with the inductive equivalence relation defined in Section 3.5.4; propositional equality would be too restrictive, since that does not allow us prove equivalence of programs using equations of the first-order effect(s) that we elaborate into.

```
Respectsᴴ : (_≈_ : ∀ {A} → Free Δ A → Free Δ A → Set₁)
          → Algᴴ H (Free Δ) → Equationᴴ H → Set₁
Respectsᴴ _≈_ alg eq =
  ∀ {vs γ} →   cataᴴ pure alg (lhs eq vs γ)
             ≈ cataᴴ pure alg (rhs eq vs γ)
```

Since elaborations are composed in parallel, the use of necessity in the defintion of equations has additional consequences for the definiton of elaboration correctness. That is, correctness of an elaboration is defined with respect to a theory whose equations have left-hand and right-hand sides that may contain term metavariables that range over programs with more higher-order effects than those the elaboration is defined for. Therefore, to state correctness, we must also close over all possible ways these additional effects are elaborated. For this, we define the following binary relation on extensible elaborations.

```
record _⊑_ (e₁ : □ (Elaboration H₁) Δ₁)
           (e₂ : □ (Elaboration H₂) Δ₂) : Set₁ where
  field
    ⦃ ≲-eff   ⦄ : Δ₁ ≲ Δ₂
    ⦃ ≲ᴴ-eff  ⦄ : H₁ ≲ᴴ H₂
    preserves-cases
      : ∀ {M} (m : ⟦ H₁ ⟧ᴴ M A)
      → (e′ : ∀[ M ⇒ Free Δ₂ ])
      →     □⟨ e₁ ⟩ .alg (mapᴴ-sig (λ {x} → e′ {x}) m)
        ≡   extract e₂ .alg
              (map-sigᴴ (λ {x} → e′ {x}) (injᴴ {X = A} m))
```

A proof of the form $e_1 \sqsubseteq e_2$ witnesses that the elaboration e_1 is included in e_2. Informally, this means that e_2 may elaborate a bigger set of higher-order effects, for which it may need to refer to a bigger set of first-order effects, but for those higher-order effects that both e_1 and e_2 know how to elaborate, they should agree on how those effects are elaborated.

We then define correctness of elaborations as follows.

```
Correctᴴ :  Theoryᴴ H → Theory Δ
          → □ (Elaboration H) Δ → Set₁
Correctᴴ Th T e =
  ∀ {Δ′ H′}
  → (e′ : □ (Elaboration H′) Δ′)
  → ⦃ _ : e ⊑ e′ ⦄
  → {eq : □ Equationᴴ _}
  → eq ◂ᴴ Th
  → Respectsᴴ (_≈⟨ T ⟩_) (extract e′) □⟨ eq ⟩
```

Which is to say that an elaboration is correct with respect to a theory of the higher-order effects it elaborates (*Th*) and a theory of the first-order effects it elaborates into (T), if all possible extensions of said elaboration respect all equations of the higher-order theory, modulo the equations of the first-order theory.

Crucially, correctness of elaborations is preserved under composition of elaborations. Figure 11 shows the type of the corresponding correctness theorem in Agda; for the full details of the proof we refer to the Agda formalization accompanying this paper [Van der Rest and Bach Poulsen, 2024]. We remark that correctness of a composed elaboration is defined with respect to the composition of the theories of the first-order effects that the respective elaborations use. Constructing a handler that is correct with respect to this composed first-order effect theory is a separate concern; a solution based on *fusion* is detailed in the work by Yang and Wu [2021].

```
compose-elab-correct : {_ : Δ₁ • Δ₂ ≈ Δ }
                     → (e₁ : □ (Elaboration H₁) Δ₁)
                     → (e₂ : □ (Elaboration H₂) Δ₂)
                     → (T₁ : Theory Δ₁)
                     → (T₂ : Theory Δ₂)
                     → (Th₁ : Theoryᴴ H₁)
                     → (Th₂ : Theoryᴴ H₂)
                     → Correctᴴ Th₁ T₁ e₁
                     → Correctᴴ Th₂ T₂ e₂
                     → Correctᴴ (Th₁ [+]ᴴ Th₂) (compose-theory T₁ T₂)
                         (compose-elab e₁ e₂)
```

Figure 11: The central correctness theorem establishing that correctness of elaborations is preserved under composition

3.5.9 *Proving Correctness of Elaborations*

To illustrate how the reasoning infrastructure build up in this section can be applied to verify correctness of elaborations, we show how to verify the *catch-return* law for the elaboration eCatch defined in Section 3.3.4. First, we define the following syntax for invoking a known elaboration.

```
module Elab (e : □ (Elaboration H) Δ) where
  ℰ⟦_⟧ : Hefty H A → Free Δ A
  ℰ⟦ m ⟧ = elaborate (extract e) m
```

When opening the module *Elab*, we can use the syntax ℰ⟦ *m*⟧ for elaborating m with some known elaboration, which helps to simplify and improve readability of equational proofs for higher-order effects.

Now, to prove that eCatch is correct with respect to a higher-order theory for the Catch effect containing the *catch-return* law, we must produce a proof that the programs $\mathcal{E}$ `catch (return x) m ⟧ and $\mathcal{E}$⟦ return ⟧ are equal (in the sense of the inductive equivalence relation defined in Section 3.5.4) with respect to some first-order theory for the Throw effect. In this instance, we do not need any equations from this underlying theory to prove the equality, but sometimes it is necessary to invoke equations of the underlying first-order effects to prove correctness of an elaboration.

```
eCatchCorrect :   {T : Theory Throw}
                → Correct^H CatchTheory T eCatch
eCatchCorrect {Δ′ = Δ′} e′ { ζ } (tt , refl) {γ = x , m} =
  begin
    ℰ⟦ `catch (pure x) m ⟧
  ≈⟪ from-≡ (sym $ ζ .preserves-cases _ ℰ⟦_⟧) ⟫
    ( ♯ (given hThrow handle (pure x) $ tt))
        ≫= maybe′ pure (ℰ⟦ m ⟧ )
  ≈⟪⟫ {- By definition of hThrow -}
    (pure (just x) ≫= maybe′ pure ((ℰ⟦ m ⟧ ≫= pure)))
  ≈⟪⟫ {- By definition of ≫= -}
    ℰ⟦ pure x ⟧
  ■
  where
    open ≈-Reasoning _
    open Elab e′
```

In the Agda formalization accompanying this paper [Van der Rest and Bach Poulsen, 2024], we verify correctness of elaborations for the higher-order operations that are part of the 3MT library by Delaware et al. [2013c]. Figure 12 shows an overview of first-order and higher-order effects included in the development, and the laws which we prove about their handlers respectively elaborations.

Effect	Laws	
Throw	$\text{`throw} \gg= k \;\equiv\; k$	*bind-throw*
State	$\text{`get} \gg= \lambda\, s \to \text{`get} \gg= k\, s \;\equiv\; \text{`get} \gg= k\, s\, s$	*get-get*
	$\text{`get} \gg= \text{`put} \;\equiv\; \text{pure}\; x$	*get-put*
	$\text{`put}\; s \gg \text{`get} \;\equiv\; \text{`put}\; s \gg \text{pure}\; s$	*put-get*
	$\text{`put}\; s \gg \text{`put}\; s' \;\equiv\; \text{`put}\; s'$	*put-put*
Reader	$\text{`ask} \gg m \;\equiv\; m$	*ask-query*
	$\text{`ask} \gg= \lambda\, r \to \text{`ask} \gg= k\, r \;\equiv\; \text{`ask} \gg= \lambda\, r \to k\, r\, r$	*ask-ask*
	$m \gg= \lambda\, x \to \text{`ask} \gg= \lambda\, r \to k\, x\, r \;\equiv\; \text{`ask} \gg= \lambda\, r \to m \gg= \lambda\, x \to k\, x\, r$	*ask-bind*
LocalReader	$\text{`local}\; f\; (\text{pure}\; x) \;\equiv\; \text{pure}\; x$	*local-pure*
	$\text{`local}\; f\; (m \gg= k) \;\equiv\; \text{`local}\; f\; m \gg= \text{`local}\; f \circ k$	*local-bind*
	$\text{`local f `ask} \;\equiv\; \text{pure} \circ f$	*local-ask*
	$\text{`local}\; (f \circ g)\; m \;\equiv\; \text{`local}\; g\; (\text{`local}\; f\; m)$	*local-local*
Catch	$\text{`catch}\; (\text{pure}\; x)\; m \;\equiv\; \text{pure}\; x$	*catch-pure*
	$\text{`catch `throw}\; m \;\equiv\; m$	*catch-throw*$_1$
	$\text{`catch}\; m\; \text{`throw} \;\equiv\; m$	*catch-throw*$_2$
Lambda	$\text{`abs}\; f \gg= \lambda\, f' \to \text{`app}\; f'\; m \;\equiv\; m \gg= f$	*beta*
	$\text{pure}\; f \;\equiv\; \text{`abs}\; (\lambda\, x \to \text{`app}\; f\; (\text{`var}\; x))$	*eta*

Figure 12: Overview of effects, their operations, and verified laws in the Agda code

3.6 RELATED WORK

As stated in the introduction of this paper, defining abstractions for programming constructs with side effects is a research question with a long and rich history, which we briefly summarize here. Moggi [1990] introduced monads as a means of modeling side effects and structuring programs with side effects; an idea which Wadler [1992] helped popularize. A problem with monads is that they do not naturally compose. A range of different solutions have been developed to address this issue [Jr., 1994, Jones and Duponcheel, 1993, Filinski, 1999, Cenciarelli and Moggi, 1993]. Of these solutions, monad transformers [Cenciarelli and Moggi, 1993, Liang et al., 1995b, Jaskelioff, 2008] is the more widely adopted solution. However, more recently, algebraic effects [Plotkin and Power, 2002] was proposed as an alternative solution which offers some modularity benefits over monads and monad transformers. In particular, whereas monads and monad transformers may “leak” information about the implementation of operations, algebraic effects enforce a strict separation between the interface and implementation of operations. Furthermore, monad transformers commonly require glue code to “lift” operations between layers of monad transformer stacks. While the latter problem is addressed by the Monatron framework of Jaskelioff [2008], algebraic effects have a simple composition semantics that does not require intricate liftings.

However, some effects, such as exception catching, did not fit into the framework of algebraic effects. *Effect handlers* [Plotkin and Pretnar, 2009b] were introduced to address this problem. Algebraic effects and handlers has since been gaining traction as a framework for modeling and structuring programs with side effects in a modular way. Several libraries have been developed based on the idea such as *Handlers in Action* [Kammar et al., 2013], the freer monad [Kiselyov and Ishii, 2015], or Idris’ `Effects` DSL [Brady, 2013a]; but also standalone languages such as Eff [Bauer and Pretnar, 2015], Koka [Leijen, 2017], Frank [Convent et al., 2020], and Effekt [Brachthäuser et al., 2020].[54]

[54] *A more extensive list of applications and frameworks can be found in Jeremy Yallop’s Effects Bibliography:* `https://github.com/yallop/effects-bibliography`

As discussed in Section 3.1.2 and Section 3.2.5, some modularity benefits of algebraic effects and handlers do not carry over to higher-order effects. Scoped effects and handlers [Wu et al., 2014, Piróg et al., 2018, Yang et al., 2022] address this shortcoming for *scoped operations*, as we summarized in Section 3.2.6. This paper provides a different solution to the modularity problem with higher-order effects. Our solution is to provide modular elaborations of higher-order effects into more primitive effects and handlers. We can, in theory, encode any effect in terms of algebraic effects and handlers. However, for some effects, the encodings may be complicated. While the complicated encodings are hidden behind a higher-order effect interface, complicated encodings may hinder understanding the operational semantics of higher-order effects, and may make it hard to verify algebraic laws about implementations of the interface. Our framework would also support elaborating higher-order effects into scoped effects and handlers, which might provide benefits for verification. We leave this as a question to explore in future work.

Although not explicitly advertised, some standalone languages, such as Frank [Convent et al., 2020] and Koka [Leijen, 2017] do have some support for higher-order effects. The denotational semantics of these features of these languages is unclear. A question for future work is whether the modular elaborations we introduce could provide a denotational model.

A recent paper by Van den Berg et al. [2021a] introduced a generalization of scoped effects that they call *latent effects* which supports a broader class of effects, including λ abstraction. While the framework appears powerful, it currently lacks a denotational model, and seems to require similar weaving glue code as scoped effects. The solution we present in this paper does not require weaving glue code, and is given by a modular but simple mapping onto algebraic effects and handlers.

Looking beyond purely functional models of semantics and effects, there are also lines of work on modular support for side effects in operational semantics [Plotkin, 2004]. Mosses' Modular Structural Operational Semantics [Mosses, 2004] (MSOS) defines small-step rules that implicitly propagate an open-ended set of *auxiliary entities* which encode common classes of effects, such as reading or emitting data, stateful mutation, and even control effects [Sculthorpe et al., 2015]. The K Framework [Rosu and Serbanuta, 2010] takes a different approach but provides many of the same benefits. These frameworks do not encapsulate operational details but instead make it notationally convenient to program (or specify semantics) with side effects.

3.7 CONCLUSION

We have presented a new solution to the modularity problem with modeling and programming with higher-order effects. Our solution allows programming against an interface of higher-order effects in a way that provides effect encapsulation, meaning we can modularly change the implementation of effects without changing programs written against the interface and without changing the definition of any interface implementations. Furthermore, the solution requires a minimal amount of glue code to compose language definitions.

We have shown that the framework supports modular reasoning on a par with algebraic effects and handlers, albeit with some administrative overhead. While we have made use of Agda and dependent types throughout this paper, the framework should be portable to less dependently-typed functional languages, such as Haskell, OCaml, or Scala. An interesting direction for future work is to explore whether the framework could provide a denotational model for handling higher-order effects in standalone languages with support for effect handlers.

POSTSCRIPT

While this chapter describes an approach for modularly defining higher-order effects that is expressive enough for describing the side effects of most of the interpreters discussed in Section 2.5, the language fragment implementing polymorphic references remains beyond its capabilities. The reason is that it requires world-indexing to maintain well-formedness of references. While extending the approach to include support for that and other intrinsically-typed effects is an interesting subject for futre work, in the remainder of this thesis we will address a different problem. That is, by working in Agda we induce significant syntactic overhead into our definitions resulting from modularity. One way to deal with this overhead is to design a dedicated meta-language that includes modularity of definitions as part of its design. The second part of this thesis explores this approach.

Part II

META LANGUAGE DESIGN

4

TOWARDS A LANGUAGE FOR DEFINING REUSABLE PROGRAMMING LANGUAGE COMPONENTS

PREFACE

We begin our exploration of meta-language design for defining reusable programming language components with the language CS, wich aims to provide a convenient surface syntax for working with extensible data types. The goal of designing this language is to gain insight into the programming abstractions that a meta-language could feature for defining reusable programming language components. While the language lacks a formal definition of its type system and semantics, it does have a prototype implementation.

4.1 INTRODUCTION

Developing programming languages is a difficult and time consuming task, which requires a lot of expertise from the language designer. One way to reduce the cost of language development is to build languages from *reusable programming components*, allowing language designers to grab off-the-shelf components for common language features. This approach has the potential to make language development both cheaper and more accesbile, while producing specifications that allow us to understand the semantics of language features independent from the language they are a part of. Modern functional programming languages, however, lack support for reuse of definitions, and as a result, language components built from algebraic data types and pattern matching functions cannot be reused without modifying or copying existing code. To illustrate the kind of language components we would like to define modularly, and where current functional programming languages fall short for this purpose, we consider the implementation of a tiny expression language and its interpreter in Haskell:

```
data Expr = Lit Int | Div Expr Expr
eval :: MonadFail m ⇒ Expr → m Int
eval (Lit x)       = return x
eval (Div e1 e2) = do
  v1 ← eval e1
  v2 ← eval e2
  if v2 ≢ 0 then return (v1 ‘div‘ v2 else fail
```

The *Expr* data type declares an abstract syntax type with constructors for literals and division, and the function *eval* implements an interpreter for *Expr*. Importantly, *eval* needs to account for possible divisions by zero: evaluating *Div* (*Lit* 10) (*Lit* 0), for example, should safely evaluate to a result that indicates a failure, without crashing. For this reason *eval* does not produce an *Int* directly, but rather wraps its result in an abstract monad *m* that encapsulates the side effects of interpretation. In this case, we only assume that *m* is a member of the *MonadFail* typeclass. The *MonadFail* class hase one function, *fail*:

```
class MonadFail m where
  fail :: m a
```

We refer to functions, such as *fail*, that allow us to interact with an abstract monad as *operations*. We choose to factor language definitions this way, because it allows us to both define a completely new interpretation such as pretty printing or compilation for *Expr* by writing new functions *pretty* :: *Expr* → *String* or *compile* :: *Expr* → *m* [*Instr*], while also having the option to change the implementation of existing semantics, by supplying alternative implementations for the fail operation. We can summarize this approach to defining language components with the following pipeline:

$$\text{Syntax} \xrightarrow{\text{denotation}} \text{Operations} \xrightarrow{\text{implementation}} \text{Result}$$

That is, a *denotation* maps *syntax* to an appropriate domain. In the definition of this domain, we distinguish between the type of the resulting value, and the side effects of computing this result, which are encapsulated in an abstract monad. We interact with this abstract monad using *operations*, and thus to extract a result we must supply a monad that *implements* all required operations.

What if we want to extend this language? To add new constructors to the abstract syntax tree, we must extend the definition of *Expr*, and modify *all* functions that match on *Expr* accordingly. Furthermore, the new clauses for these constructors may impose additional requirements on *m* for which we would need to add more typeclass constraints, and any existing instantiations of *m* would need to be updated to ensure that they are still a member of all required typeclasses.

Clearly, for these reasons *Expr* and *eval* in their current form do not work very well as a reusable language component. We introduce CS, a functional meta-language for defining reusable programming language components. The goal of CS is to provide a language in which one can define language components by defining data types and pattern matching functions, like *Expr* and *eval*, in such a way that we compose the syntax, interpretations, and effects of a language component without affecting existing defintion. Importantly, we should also retain the possibility to add completely new interpretations for existing syntax by writing a new pattern matching function. In other words, CS should solve the *expression problem* [Wadler, 1998].

We can summarize this with following concrete design goals. In CS, one should be able to

- extend existing abstract syntax types with new constructors without having to modify existing definitions,
- extend existing denotations with clauses for new constructors, and define new semantics for existing syntax by defining new denotations,
- define abstract effect operations, and use these operations to implement denotation clauses without having to worry about the operations needed by other clauses, and
- define implementations for effect operations that are independent from the implementations of other operations.

There exist abstractions, such as *Data Types à la Carte* [Swierstra, 2008] and *Algebraic Effects and Handlers* [Plotkin and Pretnar, 2009a], that achieve the same goals. These provide the well-understood formalism on wich CS is built. CS then provides a convenient surface syntax for working with these abstractions that avoids the overhead that occurs when encoding them in a host language like Haskell.

CS is work in progress. There is a prototype implementation of an interpreter and interactive programming environment which we can use to define and run the examples from this abstract. We are, however, still in the process of developing and implementing a type system. In particular, we should statically detect errors resulting from missing implementations of function clauses.

The name CS is an abbreviation of “CompositionalSemantics”. It is also the initials of Christopher Strachey, whose pioneering work [Strachey, 1966] initiated the development of denotational semantics. In *Fundamental Concepts in Programming Languages*, Strachey [2000] wrote that “the urgent task in programming languages is to explore the field of semantic possibilities”, and that we need to “recognize and isolate the central concepts” of programming languages. Today, five decades later, the words still ring true. The CS language aims to address this urgent task in programming languages, by supporting the definition of reusable (central) programming language concepts, via compositional denotation functions that map the syntax of programming languages to their meaning.

4.2 CS BY EXAMPLE

In this section, we give an example-driven introduction to CS’s features.

4.2.1 *Data Types and Functions*

CS is a functional programming language, and comes equipped with algebraic data types and pattern matching functions. We declare a new inductive data type for natural numbers as follows:

$\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Suc}\ \mathit{Nat}$

We can write functions over inductive data types by pattern matching, using a "pipe" (|) symbol to separate clauses:

$$\begin{array}{l}
\textbf{fun}\ \mathit{double} : \mathit{Nat} \rightarrow \mathit{Nat}\ \textbf{where} \\
\mid \mathit{Zero} \qquad\ \mapsto \mathit{Zero} \\
\mid (\mathit{Suc}\ n) \mapsto \mathit{Suc}\ (\mathit{Suc}\ (\mathit{double}\ n))
\end{array}$$

Not all types are user-declared: CS also offers built-in types and syntax for integers, lists, tuples, and strings.

$$\begin{array}{l}
\textbf{fun}\ \mathit{length} : \mathit{List}\ a \rightarrow \mathit{Int}\ \textbf{where} \\
\mid [\,] \qquad\quad \mapsto 0 \\
\mid (\_ :: \mathit{xs}) \ \mapsto 1 + \mathit{length}\ \mathit{xs} \\
\textbf{fun}\ \mathit{zip} : \mathit{List}\ a \rightarrow \mathit{List}\ b \rightarrow \mathit{List}\ (a * b)\ \textbf{where} \\
\mid [\,] \qquad\quad\ \ \_ \qquad\quad \mapsto [\,] \\
\mid (x :: \mathit{xs})\ (y :: \mathit{ys}) \mapsto (x, y) :: \mathit{zip}\ \mathit{xs}\ \mathit{ys}
\end{array}$$

Both *length* and *zip* are polymorphic in the type of elements stored in the input list(s). Functions implicitly generalize over any free variables in their type signature.

4.2.2 *Effects and Handlers*

CS supports effectful programs by means of effects and handlers in the spirit of Plotkin and Pretnar [2009a], adapted to support higher-order operations. The key idea of the effects-and-handlers approach is to declare the syntax of effectful *operations*, and assign a semantics to these operations in a separate *handler*. Programs compute values and have side effects, and operations act as the interface through which these side effects are triggered.

We declare a new effect *Fail* with a single operation fail in CS as follows:

$$\begin{array}{l}
\textbf{effect}\ \mathit{Fail}\ \textbf{where} \\
\quad \mathsf{fail} : \{[\mathit{Fail}]\ a\}
\end{array}$$

Effects in CS are declared with the **effect** keyword, and we declare its operations by giving a list of GADT-style signatures. In this case, the fail operation is declared to have type $\{[\mathit{Fail}]\ a\}$. We enclose the type of fail in braces ({−}) to indicate that the name fail refers to a *suspended computation* (Section 4.2.3). Suspended computations are annotated with an *effect row*, enclosed in square brackets ([−]), denoting the side effects of running the computation. Invoking the fail operation has *Fail* as a side effect.

We can use the *Fail* effect to implement a safe division function that invokes fail if the divisor is zero.

```
fun safeDiv : Int → Int → [Fail] Int where
  | x 0 ↦ fail!
  | x y ↦ ...
```

The postfix exclamation mark to fail is necessary to force the suspended computation. Here, we want to refer to the "action" of failing, rather than the computation itself, following Frank's [Convent et al., 2020] separation between "being and doing". We elaborate on this distinction in Section 4.2.3.

A function's type signature must explicitly indicate its side effects. In this case, we annotate the return type of *safeDiv* with the *Fail* effect to indicate that its implementation uses operations of this effect. Removing the annotation would make the above invocation of fail ill-typed. For functions that have no side effects, we may omit its row annotation: $a \to b$ is synonymous to $a \to [\,]\ b$

Handlers discharge an effect from annotations by assigning a semantics to its operations. For the *Fail* effect, we can do this by encoding exceptions in the *Maybe* type.

```
data Maybe a = Just a | Nothing
handler hFail : {[Fail|e] a} → {[e] (Maybe a)} where
  | fail     k ↦ {Nothing}
  | return x ↦ {Just x}
```

The handler *hFail* takes a value annotated with the *Fail* effect, and produces a *Maybe* value annotated with the remaining effects e. All free effect row variables in a signature, like e, are implicitly generalized over. When defining a handler we must provide a clause for each operation of the handled effect. Additionally, we must write a **return** clause that lifts pure values into the domain that encodes the effect's semantics. Operation clauses have a *continuation parameter*, k, with type $b \to [e]\ (Maybe\ a)$, which captures the remainder of the program starting from the current operation. Handlers may use the continuation parameter to decide how execution should resume after the current operation is handled. For example, when handing the fail operation we terminate execution of the program by ignoring this continuation.

We use the continuation parameter in a different way when defining a handler for a *State* effect, where $s : Set$ is a parameter of the module in which we define the effect.

```
effect State where
  | get :       [State] s
  | put : s → [State] ()
handler hState : {[State|e] a} → s → {[e] (a ∗ s)} where
  | get         st k ↦ k st st
  | (put st′) st k ↦ k () st′
  | return x st     ↦ {(x, st)}
```

For both the get and put operations, we use the continuation parameter k to implement the corresponding branch in *hState*. The continuation expects a value whose type corresponds to the return type of the current operation, and produces a computation with the same type as the return type of the handler. For the put operation, for example, this means that k is of type $() \to s \to \{[e]\ (a * s)\}$. The implementation of *hState* for get and put then simply invokes k, using the current state as both the value and input state (get), or giving a unit value and using the given state st' as the input state (put). Effectively, this means that after handling get or put, execution of the program resumes, respectively with the same state or an updated state st'.

Handlers in CS are so-called *deep handlers*, meaning that they are automatically distributed over continuations. For the example above, this means that that the *State* effect is already handled in the computation returned by k. The alternative is *shallow handlers*, in which case k would return a computation of type $\{[State|e]\ a\}$. When using shallow handlers, the programmer is responsible for recursively applying the handler to the result of continuations.

While shallow handlers are more expressive, unlike deep handlers they are not guaranteed to correspond to a fold over the effect tree, meaning that they are potentially harder to reason about.

4.2.3 *Order of Evaluation, Suspension, and Enactment*

Inspired by Frank [Convent et al., 2020], CS allows effectful computations to be used as if they were pure values, without having to sequence them. Sub-expressions in CS are evaluated from left to right, and the side effects of computational subexpressions are evaluated eagerly in that order. For example, consider the following program:

```
fun f : Int → [Fail] Int where
 | n ↦ fail! + n
```

Here, we use the expression fail! (whose type is instantiated to $[Fail]\ Int$) as the first argument to $+$, where a value of type Int is expected. This is fine, because side effects that occur during evaluation of sub-terms are discharged to the surrounding context. That is, the side effects of evaluating computational sub-terms in the definition of f become side effects of f itself.

In practice, this means that function application in CS is not unlike programming in an *applicative style* in Haskell. For instance, when using the previously-defined handler $hFail$, which maps the $Fail$ effect to a $Maybe$, we can informally understand the semantics of the CS program above as equivalent to the following Haskell program:

```
f :: Int → Maybe Int
f n = (+) <$> Nothing <*> pure n
```

Equivalently, we could write the following *monadic* program in Haskell, which makes the evaluation order explicit.

```
f :: Int → Maybe Int
f n = do x ← Nothing
         y ← pure n
         return (x + y)
```

CS's eager treatment of side effects means that effectful computations are not first-class values, in the sense that we cannot refer to an effectful computation without triggering its side effects. To treat computations as first-class, we must explicitly *suspend* their effects using *braces*:

```
fun f′ : Int → {[Fail] Int} where
| n ↦ {f n}
```

The function f' is no longer a function on *Int* that may fail, but instead a function from *Int* to a computation that returns and *Int*, but that could also fail. We indicate a suspended computation in types using braces ({/}), and construct a suspension at the term level using the same notation.

To *enact* the side effects of a suspended computation, we postfix it with an exclamation mark (!). For example, the expression $(f'\ 0)!$ has type $[Fail]\ Int$, whereas the expression $(f'\ 0)$ has type $\{[Fail]\ Int\}$. We see the same distinction with operations declared using the **effect** keyword. When we write fail, we refer to the operation in a descriptive sense, and we can treat it like any other value without having to worry about its side effects. When writing fail! , on the other hand, we are really performing the action of abruptly terminating: fail *is* and fail! *does*.

4.2.4 *Modules and Imports*

CS programs are organized using modules. Modules are delimited using the **module** and **end** keywords, and their definitions can be brought into scope elsewhere using the **import** keyword. All declarations—i.e., data types, functions, effects, and handlers—must occur inside a module.

```
module A where
  fun f : Int → Int where
  | n ↦ n + n
end
module B where
  import A
```

```
    fun g : Int → Int where
     | n ↦ f n
  end
```

In addition to being an organizational tool, modules play a key role in defining and composing modular data types and functions.

4.2.5 *Composable Data Types and Functions*

In addition to plain algebraic data types and pattern matching functions, declared using the **data** and **fun** keywords, CS also supports case-by-case definitions of *extensible* data types and functions. In effect, CS provides a convenient surface syntax for working with DTC-style [Swierstra, 2008] definitions, which relies on an embedding of the initial algebra semantics [Goguen, 1976, Johann and Ghani, 2007] of data types to give a semantics to extensible and composable algebraic data types and functions, meaning that extensible functions have to correspond to a *fold* [Meijer et al., 1991]. In CS, one can program with extensible data types and functions in the same familiar way as with their plain, non-extensible counterparts.

The module system plays an essential role in the definition of composable data types and functions. That is, modules can inhabit a *signature* that declares the extensible types and functions for which that module can give a partial definition. In a signature declaration, we use the keyword **sort** to declare an extensible data type, and the **alg** keyword to declare an extensible function, or *algebra*. By requiring extensible functions to be defined as algebras over the functor semantics of extensible data types, we enforce *by construction* that they correspond to a fold.

As an example, consider the following signature that declares an extensible data type *Expr*, which can be evaluated to an integer using *eval*.

```
  signature Eval where
    sort Expr : Set
  alg eval : Expr → Int
end
```

To give cases for *Expr* and *eval*, we define modules that inhabit the *Eval* signature.

```
module Lit : Eval where
  cons Lit : Int → Expr
  case eval (Lit x) ↦ x
end
module Add : Eval where
  cons Add : Expr → Expr → Expr
  case eval (Add x y) ↦ x + y
end
```

The **cons** keyword declares a new constructor for an extensible data type, where we declare any arguments by giving a GADT-style type signature. We declare clauses for functions that match on an extensible type using the **case** keyword. For every newly declared constructor of an extensible data type, we have an obligation to supply exactly one corresponding clause *for every extensible function that matches on that type*. CS has a coverage checker that checks whether modules indeed contain all necessary definitions, in order to rule out partiality resulting from missing patterns. For example, omitting the *eval* case from either the module *Lit* or *Add* above will result in a static error. Coverage is checked locally in modules, and preserved when composing signature instances.

In the definition of *eval* in the module *Add*, we see the implications of defining function clauses as algebras. We do not have direct control over recursive calls to *eval*. Instead, in **case** declarations, any recursive arguments to the matched constructor are replaced with the result of recursively invoking *eval* on them. In this case, this implies that x and y do not refer to expressions. Rather, if we invoke *eval* on the expression *Add e1 e2*, in the corresponding **case** declaration, x and y are bound to *eval e1* and *eval e2* respectively. We could encode the same example in Haskell as follows, but to use *eval* on concrete expressions additionally requires explicit definitions of a type level fixpoint and fold operation. In CS, this encoding layer is hidden by the language.

```
data Add e = Add e e
eval :: Add Int → Int
eval (Add x y) = x + y
```

To compose signature instances we merely have to import them from the same location.

```
module Program where
  import Lit, Add
  -- Evaluates to 3
  fun test : Int = eval (Add (Lit 1) (Lit 2))
end
```

By importing both the *Lit* and *Add* modules, the names *Expr* and *eval* will refer to the composition of the constructors/clauses defined in the imported signature instances. Here, this means that we can construct and evaluate expressions that consist of both literals and addition. Furthermore, to add a new constructor into the mix, we can simply define a new module that instantiates the *Eval* signature, and add it to the import list.

To define an alternative interpretation for *Expr*, we declare a new signature. In order to reference the **sort** declaration for *Expr*, we must import the *Eval* signature.

```
signature Pretty where
  import Eval -- brings 'Expr' into scope
  alg pretty : Expr → String
end
```

We declare cases for *pretty* by instantiating the newly defined signature, adding **import** declarations to bring relevant **cons** declaration into scope.

```
module PrettyAdd : Pretty where
  import Add -- brings 'Add' into scope
  case pretty (Add s1 s2) = s1 ++ " + " ++ s2
end
```

It is possible for two modules to be *conflicting*, in the sense that they both define an algebra case for the same constructor. This would happen, for example, if we were to define another module *PrettyAdd2* that also implements *pretty* for the constructor *Add*. Importing two conflicting modules should result in a type error, since the semantics of their composition is unclear.

4.3 DEFINING REUSABLE LANGUAGE COMPONENTS IN CS

In this section, we demonstrate how to use the features of CS introduced in the previous section to define reusable language components. We work towards defining a reusable component for function abstraction, which can be composed with other constructs, and for which we can define alternative implementations. As an example, we will show that we can use the same component defining functions with both a call-by-value and call-by-name strategy.

4.3.1 *A Signature for Reusable Components*

The first step is to define an appropriate module signature. We follow the same setup as for the *Eval* signature in Section 4.2.5. That is, we declare an extensible sort *Expr*, together with an algebra *eval* that consumes values of type *Expr*. The result of evaluation is a *Value*, with potential side effects *e*. The side effects are still abstract in the signature definition. The effect variable *e* is universally quantified, but instantiations of the signature *Eval* can impose additional constraints depending on which effects they need to implement *eval*. As a result, when invoking *eval*, *e* can be instantiated with any effect row that satisfies all the constraints imposed by the instances of *Eval* that are in scope.

```
signature Eval where
  sort Expr : Set
  alg  eval  : Expr → {[e] Value}
end
```

We will consider the precise definition of *Value* later in Section 4.3.3. For now, it is enough to know that it has a constructor $Num : Int \rightarrow Value$ that constructs a value from an integer literal.

4.3.2 *A Language Component for Arithmetic Expressions*

Let us start by defining instances of the *Eval* signature for the expression language from the introduction. First, we define a module for integer literals.

```
module Lit : Eval where
  cons Lit : Int → Expr
  case eval (Lit n) = {Num n}
end
```

The corresponding clause for *eval* simply returns the value *n* stored inside the constructor. Because the interpreter expects that we return a suspended computation, we must wrap *n* in a suspension, even though it is a pure value. Enacting this suspension, however, does not trigger any side effects, and as such importing *Lit* imposes no constraints on the effect row *e*.

Next, we define a module *Div* that implements integer division.

```
module Div : Eval where
  cons Div : Expr → Expr → Expr
  case eval (Div m1 m2) = {safeDiv m1! m2! }
```

Looking at the implementation of *eval* in the module *Div* we notice two things. First, the recursive arguments to *Div* have been replaced by the result of calling *eval* on them, meaning that *m1* and *m2* are now *computations* with type $\{[e]\ Int\}$, and hence we must use enactment before we can pass the result to *safeDiv*. Enacting these computations may trigger side effects, so the order in which sub-expressions are evaluated determines in which order these side effects occur in the case that expressions contain more than one enactment. Sub-expressions in CS are evaluated from left to right. Second, the implementation uses the function *safeDiv*, defined in Section 4.2.2, which guards against errors resulting from division by zero.

The function *safeDiv* is annotated with the *Fail* effect, which supplies the fail operation. By invoking *safeDiv* in the defintion of *eval*, which from the definition of *Eval* has type $Expr \rightarrow \{[e]\ Int\}$, we are imposing a constraint on the effect row e that it contains at least the *Fail* effect. In other words, whenever we import the module *Div* we have to make sure that we instantiate e with a row that has *Fail* in it. Consequently, before we can extract a value from any instantiation of *Eval* that includes *Div*, we must apply a handler for the *Fail* effect.

Since the interpreter now returns a *Value* instead of an *Int*, we must modify *safeDiv* accordingly. In practice this means that we must check if its arguments are constructed using the *Num* constructor before further processing the input. Since *safeDiv* already has *Fail* as a side effect, we can invoke the fail operation in case an argument was constructed using a different constructor than *Num*.

4.3.3 *Implementing Functions as a Reusable Effect*

CS's effect system can describe much more sophisticated effects than *Fail*. The effect system permits fine-grained control over the semantics of operations that affect a program's control flow, even in the presence of other effects. To illustrate its expressiveness, we will now consider how to define function abstraction as a reusable effect, and implement two different handlers for this effect corresponding to a call-by-value and call-by-name semantics. Implementing function abstraction as an effect is especially challenging since execution of the function body is deferred until the function is applied. From a handler's perspective, this means that the function body and its side effect have to be postponed until a point beyond its own control, a pattern that is very difficult to capture using traditional algebraic effects.

We will see shortly how CS addresses this challenge. A key part of the solution is the ability to define *higher-order operations*: operations with arguments that are themselves effectful computations, leaving it up to the operation's handler to enact the side effects of higher-order arguments. The *Fun* effect, which implements function abstraction, has several higher-order operations.

```
effect Fun where
 | lam   : String → {[Fun] Value} → {[Fun] Value}
 | app   : Value  → Value         → {[Fun] Value}
 | var   : String                 → {[Fun] Value}
 | thunk : {[Fun] Value}          → {[Fun] Value}
```

The *Fun* effect defines four operations, three of which correspond to the usual constructs of the λ-calculus. The thunk operation has no counterpart in the λ-calculus, and postpones evaluation of a computation. It is necessary for evaluation to support both a call-by-value and call-by-name evaluation strategy

When looking at the lam and thunk operations, we find that they both have parameters annotated with the *Fun* effect. This annotation indicates that they are higher-order parameters. By allowing higher-order parameters to operations, effects in CS do not correspond directly to algebraic effects. Instead, to give as semantics to effects in CS, we must use a flavor of effects that permits higher-order syntax, such as *Latent Effects* [Van den Berg et al., 2021a].

As a result, any effects of the computations stored in a closure or thunk are postponed, leaving it up to the handler to decide when these take place.

USING THE *Fun* EFFECT To build a language with function abstractions that uses the *Fun* effect, we give an instance of the *Eval* signature that defines the constructors *Abs*, *App*, and *Var* for *Expr*. We extend *eval* for these constructors by mapping onto the corresponding operation.

```
module Lambda : Eval where
  cons Abs : String → Expr → Expr
   |   App : Expr   → Expr → Expr
   |   Var : String        → Expr
  case eval (Abs x m)     = lam x m
   |   eval (App m1 m2) = app m1! (thunk m2)!
```

```
    |   eval (Var x)        = var x
end
```

Crucially, in the case for *Abs* we pass the effect-annotated body *m*, with type {[*e*] *Value*}, to the lam operation directly without extracting a pure value first. This prevents any effects in the body of a lambda from being enacted at the definition site, and instead leaves the decision of when these effects should take place to the used handler for the *Fun* effect. Similarly, in the case for *App*, we pass the function argument *m2* to the thunk operation directly, postponing any side effects until we force the constructed thunk. The precise moment at which we will force the thunk constructed for function arguments will depend on whether we employ a call-by-value or call-by-name strategy. We must, however, enact the side effects of evaluating the function itself (i.e., *m1*), because the app operation expects its arguments to be a pure value.

We implement call-by-value and call-by-name handlers for *Fun* in a new module, which also defines the type of values, *Value*, for our language. To keep the exposition simple, *Value* is not an extensible **sort**, but it is possible to do so in CS.

Values in this language are either numbers (*Num*), function closures (*Clo*), or thunked computations (*Thunk*). We define the type of values together in the same module as the handler(s) for the *Fun* effect. This module is parameterized over an effect row *e*, that denotes the *remaining effects* that are left after handling the *Fun* effect. In this case, *e* is a module parameter to express that the remaining effects in the handlers that we will define coincide with the effect annotations of the computations stored in the *Clo* and *Thunk* constructors, allowing us to run these computations in the handler.

```
module HLambda (e : Effects) where
  import Fun
  type Env   = List (String * Value)
  data Value = Num Int
             | Clo String (Env → {[Fail|e] Value}) Env
             | Thunk ({[Fail|e] Value})
```

```
handler hCBV  : {[Fun|e] Value}
                  → Env → {[Fail|e] Value} where
  | (lam x f)                     nv k ↦ k (Clo x f nv) nv
  | (app (Clo x f nv′) (Thunk t)) nv k ↦ k (f ((x, t!) :: nv′))! nv
  | (app _ _)                     _  _ ↦ {fail! }
  | (var x)                       nv k ↦ k (lookup nv x)! nv
  | (thunk f)                     nv k ↦ k (Thunk {f nv}) nv
  | return v                      nv   ↦ {v}
```

```
    -- ... (handlers for the Fun effect) ...
  end
```

Figure 13: A Handler for the *Fun* effect, implementing a call-by-value semantics for function arguments. The gray highlights indicate where thunks constructed for function arguments are forced.

CALL-BY-VALUE We are now ready to define a handler for the *Fun* effect that implements a call-by-value evaluation strategy. Figure 13 shows its implementation.

The **return** case is unremarkable: we simply ignore the environment *nv* and return the value *v*. The cases for lam and thunk are similar, as in both cases we do not enact the side effects associated with the stored computation *f*, but instead wrap this computation in a *Closure* or *Thunk* which is passed to the continuation *k*. For variables, we resolve the identifier *x* in the environment and pass the result to the continuation.

A call-by-value semantics arises from the implementation of the app case. The highlights (e.g., t!) indicate where the thunk we constructed for the function argument in *eval* is forced. In this case, we force this argument thunk immediately when encountering a function application, meaning that any side effects of the argument take place *before* we evaluate the function body.

CALL-BY-NAME The handler in Figure 14 shows an implementation of a call-by-name semantics for the *Fun* effect. The only case that differ from the call-by-value handler in Figure 13 are the app and var cases.

```
handler hCBN  : {[Fun|e] Value}
                → Env → {[Fail|e] Value} where
| (lam x f)             nv k ↦ k (Clo x f nv) nv
| (app (Clo x f nv′) v) nv k ↦ k (f ((x, v) :: nv′))! nv
| (app _ _)             _  _ ↦ {fail! }
| (var x)               nv k ↦ match (lookup x nv)! with
                                 | ( Thunk t ) ↦ k  t!  nv
                                 | v           ↦ k v nv
                               end
| (thunk f)             nv k ↦ k (Thunk {f nv}) nv
| return v              nv   ↦ {v}
```

Figure 14: A Handler for the *Fun* effect, implementing a call-by-name semantics for function arguments. The gray highlights indicate where thunks constructed for function arguments are forced.

In the case for app, we now put the argument thunk in the environment immediately, without forcing it first. Instead, in the case for var, we check if the variable we look up in the environment is a *Thunk*. If so, we force it and pass the resulting value to the continuation. In effect, this means that for a variable that binds an effectful computation, the associated side effects take place every time we use that variable, but not until we reference it for the first time.

4.3.4 *Example Usage*

To illustrate how to use the reusable components defined in this section, and the difference between the semantics implemented by *hCBV* (Figure 13) and *hCBN* (Figure 14), we combine the *Lambda* module with the modules for *Div* and *Lit*. Figure 15 shows the example.

When importing the modules *HLambdaCBV* and *HLambdaCBN*, we pass an explicit effect row that corresponds to the effects that remain after handling the *Fun* effect. Because we handle *Fun* after handling the *Fail* effect introduced by *Div*, we pass the empty row. To evaluate expressions, we have to invoke *hFail* twice: first to handle the instance of the *Fail* effect introduced by *eval* for the *Div*

```
module Test where
  import Prelude
         , Fun
         , Fail
         , HLambdaCBV []
         , HLambdaCBN []
         , Lambda
         , Lit
         , Div
  fun execCBV : Expr → Maybe (Maybe Value) where
   | e ↦ (hFail (hCBV (hFail (eval e)) []))!
  fun execCBN : Expr → Maybe (Maybe Value) where
   | e ↦ (hFail (hCBN (hFail (eval e)) []))!
  fun expr : Expr = App (Abs "x" (Lit 10)) (Div (Lit 5) (Lit 0))
   -- evaluates to Just Nothing
  fun result1 : Maybe (Maybe Value) = execCBV expr
   -- evaluates to Just (Just (Num 10))
  fun result2 : Maybe (Maybe Value) = execCBN expr
end
```

Figure 15: Examples of different outcomes when using a call-by-value or call-by-name evaluation strategy

constructor, and later to handle the *Fail* instance introduced by applying *hCBV*/*hCBN*. Consequently, the result of evaluating is a nested *Maybe*, where the inner instance indicates errors resulting from division by zero, and the outer instance errors thrown by the handler. Evaluating *result1* and *result2* shows the difference between using the call-by-value and call-by-name semantics for functions.

4.4 RELATED WORK

4.4.1 *Effect Semantics*

Monads, originally introduced by Moggi [1991] have long been the dominant approach to modelling programs with side effects. They are, however, famously hard to compose, leading to the development of *monad transformers* [Liang et al., 1995a] as a technique for building monads from individual definitions of effects. *Algebraic effects* [Plotkin and Power, 2003] provide a more structured approach towards this goal, where an effect is specified in terms of the *operations* that we can use to interact with it. The behaviour of these operations is governed by a set of equations that specify its well-behavedness. Later, Plotkin and Pretnar [Plotkin and Pretnar, 2009a] extended the approach with *handlers*, which define interpretations of effectful operations by defining a homomorphism from a *free model* that trivially inhabits the equational theory (i.e., syntax) to a programmer-defined domain, making the approach attractive for implementing effects as well. Perhaps the most well-known implementation of algebraic effects and handlers is the *free monad* [Kammar et al., 2013], and this implementation is often taken as the semantic foundation of languages with support for effect handlers. Schrijvers et al. [2019] showed that algebraic effects implemented using the free monad correspond to a sub-class of monad-transformers. The algebraic effects and handlers approach provides a solid formal framework for understanding effectful programs in which we intend to ground CS′ semantics of effects and handlers.

A crucial difference between CS′ effects and handlers, and the original formulation by Plotkin and Pretnar [2009a], is the support for *higher-order operations*. Although it is possible to implement such operations in algebraic effects by defining them as handlers, this breaches the separation between the syntax and semantics of effects that underpins CS′ design. *Scoped Effects* [Wu et al., 2014] were proposed as an alternative flavor of algebraic effects that supports higher-order syntax, recovering a separation between the syntax semantics of effects for higher-order operations. In subsequent work, Piróg et al. [2018] adapted the categorical formulation of algebraic effects to give Scoped Effects a similar formal underpinning. Unfortunately, Scoped Effects is not suitable out-of-the-box as a model for effects and handlers in CS, because it cannot readily capture operations that arbitrarily postpone the execution of their higher-order arguments, such as lam. *Latent effects* were developed by Van den Berg et al. [2021a] as a refinement of scoped effects that solves this issue. Key to their approach is a *latent effect functor*, which explicitly tracks semantic residue of previously-installed handlers, allowing for a more fine-grained specification of the types of the computational arguments of operations. With Latent Effects, it is possible to capture function abstraction as a higher-order operation. It remains future work to formulate a precise model of effectful computation for CS, and to establish if and how CS' effect handlers correspond to Latent Effects.

4.4.2 *Implementations of Algebraic Effects and Handlers*

There are many languages with support for algebraic effects and handlers. Perhaps the most mature is Koka [Leijen, 2017], which features a Hindley/Milner-style row-polymorphic type system. While we borrow from Frank [Convent et al., 2020] a CBPV-inspired [Levy, 2004] distinction between computations and values, Koka is purely call-by-value, and only functions can be effectful. Frank [Convent et al., 2020], on the other hand, does maintain this distinction between values and computations. Its type system relies on an *ambient ability* and implicit row polymorphism to approximate effects. Handlers are not first-class constructs in Frank. Instead, functions may adjust the ambient ability of their arguments by specifying the behaviour of operations. This provides some additional flexibility over built-in handers, for example by permitting *multihandlers* that handle multiple effects at once. Both Koka and Frank lack native support for higher order effects, thus higher-order operations must be encoded in terms of handlers. This means that it is not possible to define higher order operations while maintaining the aforementioned distinction between the syntax and semantics of effects.

Eff [Bauer and Pretnar, 2015] is a functional language with support for algebraic effects and handlers, with the possibility to dynamically generate new operations and effects. In later work, Bauer and Pretnar [2014] developed a type-and-effect system for Eff, together with an inference algorithm [Pretnar, 2014]. The language Links [Lindley and Cheney, 2012] employs row-typed algebraic effects in the context of database programming. Their system is based on System F extended with effect rows and row polymorphism, and limits effectful computations to functions similar to Koka. Importantly, their system tracks effects using Rémy-style rows [Rémy, 1989], maintaining so-called *presence types* that can additionally express an effect's absence from a computation. Brachthäuser et al. [2020] presented *Effekt* as a more practical implementation of effects and handlers, using *capability based* type system where effect types express a set of capabilities that a computation requires from its context.

4.4.3 *Semantics of Composable Data Types and Functions*

We give a semantics to extensible data types and functions in CS using the initial algebra semantics [Goguen, 1976, Johann and Ghani, 2007] of an underlying signature functor. *Data Types à la Carte* (DTC) [Swierstra, 2008] solves the expression problem in Haskell by embedding this semantics into the host language. In later work, Bahr [2014] and Bahr and Hvitved [2011, 2012a] extended the approach to improve its expressiveness and flexibility.

DTC, like any approach that relies on initial algebra semantics, limits the modular definition of functions to functions that correspond to a *fold* over the input data. While this may seem restrictive, in practice more complicated traversals can often be encoded as a fold, such as paramorphisms [Meijer et al., 1991] or some classes of attribute grammars [Johnsson, 1987]. While CS currently only has syntax for plain algebras and folds, we plan to extend the syntax for working with extensible data types and functions to accomodate a wider range of traversals in the future.

Carette et al. [2009a] showed how to define interpreters by encoding object language expressions as a term that represents their traversal. These traversals correspond to a fold, but abstract over the algebra that defines the computation, meaning that alternative semantics can be assigned to terms by picking a suitable algebra. Semantics are defined as type class instances in the host language, thus one can build object language terms in a modular way by defining multiple different type classes that correspond to different syntactical constructs.

4.4.4 *Row Types*

While a concrete design for CS′ type system is still emerging, we anticipate that it will make heavy use of *row types*, both for tracking effects and typing extensible types and functions. While to the best of our knowledge no type system exists with this combination of features, all the ingredients are there in the literature. Originally, row types were incepted as a means to model inheritance in object-oriented languages [Wand, 1989, Rémy, 1989], and later extensible records [Blume et al., 2006, Gaster and Jones, 1996]. More recently, they also gained popularity in the form of row-based effect systems with the development of languages such as Koka [Leijen, 2017] and Links [Lindley and Cheney, 2012]. Their use for typing extensible algebraic data types and pattern matching functions is less well-studied. For the most part, row types in this context exist implicitly as part of encoding techniques such as DTC [Swierstra, 2008], where we can view the use of signature functors and functor co-products as an embedding of row-typed extensible variants in the host language's type system. Various refinements of DTC [Morris, 2015, d. S. Oliveira et al., 2015, Bahr, 2014] make this connection more explicit by using type-level lists to track the composition of extensible data. A notable exception is the Rose [Morris and McKinna, 2019a] language, which has a row-based type system with built-in support for extensible variants and pattern matching function.

4.5 FUTURE WORK

CS is an ongoing research project. Here, we briefly summarize the current state, and some of the challenges that still remain.

While we can implement the examples from this paper in the current prototype implementation of CS, the language still lacks a complete specification. As such, the immediate next steps are to develop specifications of the type system and operational semantics. This requires us to address several open research questions, such as giving a semantics to the flavor of higher-order effects used by CS, and applying row types to type CS' extensible data types and functions. While the Rose language supports extensible variants, this support is limited to non-recursive types. For CS, we would need to adapt their type sytem to support recursive extensible data types as well. Designing a small core calculus into which CS can be translated could be potential way to explore these questions, making a formalization of the language in a proof assistant more attainable, by formalizing the core language. Further down the line, we also intend to explore a denotational model for effect handlers in CS, giving the language a more solid formal foundation, similar to existing programming languages based on algebraic effects and handlers.

In the future, we also hope to enforce stronger properties about specifications defined in CS through the language's type system. The prime example are *intrinsically-typed definitional interpreters* [Augustsson and Carlsson, 1999], which specify a language's operational semantics such that it is *type sound by construction*.

4.6 CONCLUSION

Reusable programming language components have the potential to significantly reduce the amount of time, effort, and expertise needed for developing programming languages. In this paper, we presented CS, a functional meta-language for defining reusable programming language components. CS enables the defintion of reusable language components using algebraic data types and pattern matching functions, by supporting *extensible data types and functions*, which are defined on a case-by-case basis. Additionally, CS features built-in support for *effects and handlers* for defining the side effects of a language. The flavor of effects and handlers implemented by CS supports *higher-order* operations, and can be used to define features that affect a program's control flow, such as function abstraction, as a reusable effect. We illustrated how these features can be used for developing reusable programming language components by defining a component for function abstraction, which can be composed with other language components and evaluated using both a call-by-value and call-by-name strategy.

POSTSCRIPT

The language design we discussed in this section provides us for a programming model for working with extensible data types. What is clearly missing is a formal specification of a typing discipline. Crucially, such a typing discipline should guarantee that all extensions of existing data are safe, in the sense that the addition of new constructors to a data type should not result in a scenario where functions over an extensible data type can be applied to inputs on which they are not defined. When embedding modular data types in functional languages using the Data Types á la Carte approach, type safe modularity is enforced by the host languages type system using a de facto shallow embedding of *row-typed algebraic data types*. This already provides us with some hints for how a suitable type system could be designed. In the next chapter, we design such a type system for a small calculus featuring primitives for programming with modular data types.

5

TYPES AND SEMANTICS FOR EXTENSIBLE DATA TYPES

PREFACE

In this chapter, we work towards providing a formal foundation for the language design proposed in Chapter 5. We do this by developing a core calculus that features primitives for working with extensible data types, together with a type system and (categorical) semantics. A potential way to connect the work from this and the previous chapter would be to define a desugaring from programs in CS to the calculus presented in this chapter, although that is at this point still future work.

5.1 INTRODUCTION

A common litmus test for a programming language's capability for modularity is whether a programmer is able to extend existing data with new ways to construct it as well as to add new functionality for this data. All in a way that preserves static type safety; a conundrum which Wadler [1998] dubbed the *expression problem*. When working in pure functional programming languages, another modularity question is how to model side effects modularly using, e.g., *monads* [Moggi, 1991]. Ideally, we would keep the specific monad used to model the effects of a program abstract and program against an *interface* of effectful operations instead, defining the syntax and implementation of such interfaces separately and in a modular fashion.

The traditional approach for tackling these modularity questions in pure functional programming languages is by embedding the *initial algebra semantics* [Goguen, 1976] of inductive data types in the language's type system. By working with such embeddings in favor of the language's built-in data types we gain modularity without sacrificing type safety. This approach was popularized by Swierstra's *Data Types à la Carte* [Swierstra, 2008] as a solution to the expression problem, where it was used to derive modular interpreters for a small expression language. In later work, similar techniques were applied to define the syntax and implementation of a large class of monads using (algebraic) effects and handlers based on different flavors of inductively defined *free monads*. This was shown to be an effective technique for modularizing both first order [Kammar et al., 2013] and higher-order [Wu et al., 2014, Bach Poulsen and Van der Rest, 2023, Van den Berg et al., 2021a] effectful computations.

The key idea that unifies these techniques is the use of *signature functors*, which act as a de facto syntactic representation of an inductive data type or inductively defined free monad. Effectively, this defines a generic inductive data type or free monad that takes its constructors as a parameter. The crucial benefit of this setup is that we can compose data types and effects by taking the coproduct of signature functors, and we can compose function cases defined over these signature functors in a similarly modular way. Inductive data types and functions in mainstream functional programming languages generally do not support these kinds of composition.

While embedding signature functors has proven itself as a tremendously useful technique for enhancing functional languages with a higher degree of type safe modularity, the approach has some downsides:

- Encodings of a data type's initial algebra semantics lacks the syntactic convenience of native data types, especially when it comes to constructing and pattern matching on values. Further overhead is introduced by their limited interoperability, which is typically relies on user-defined isomorphisms.

- The connection between initial algebra semantics encodings of data types, and the mathematical concepts that motivate them remains implicit. This has two drawbacks: (1) the programmer has to write additional code witnessing that their definitions possess the required structure (e.g., by defining instances of the `Functor` typeclass), and (2) a compiler cannot leverage the properties of this structure, such as by implementing (provably correct) optimizations based on the well-known map and fold fusion laws.

In this paper, we explore an alternative perspective by making type-safe modularity part of the language's design, by including built-in primitives for the functional programmer's modularity toolkit—e.g., functors, folds, fixpoints, etc. We believe that this approach has the potential to present the programmer with more convenient syntax for working with extensible data types (see, for example, the language design proposed by Van der Rest and Bach Poulsen [2022]). Furthermore, by supporting type-safe modularity through dedicated language primitives, we open the door for compilers to benefit from their properties, for example by applying fusion based optimizations.

5.1.1 *Contributions*

The semantics of (nested) algebraic data types has been studied extensively in the literature (e.g., by Johann et al. [2021], Johann and Polonsky [2019], Johann and Ghiorzi [2021], and Abel and Matthes [2002], Abel et al. [2003, 2005]) resulting in the development of various calculi with the purpose of studying different aspects of the semantics of programming with algebraic data types. In this paper, we build on these works to develop a core calculus that seeks to distill the essential language features needed for developing programming languages with built-in support for type-safe modularity while retaining the same formal foundations. Although the semantic ideas that we build on to develop our calculus are generally well-known, their application to improving the design of functional programming languages has yet to be explored in depth. It is still future work to leverage the insights gained by developing this calculus in the design of programming language that provide better ergonomics for working with extensible data types, but we believe the development of a core calculus capturing the essentials of programming with extensible data types to be a key step for achieving this goal. To bridge from the calculus presented in this paper to a practical language design, features such as *smart constructors*, *row types*, and *(functor) subtyping* (as employed, for example, by Morris and McKinna [2019a] and Hubers and Morris [2023]) would be essential. We make the following technical contributions:

- We show (in Section 5.2) how modular functions over algebraic data types in the style of Data Types à la Carte and modular definitions of first-order and higher-order (algebraic) effects and handlers based on inductively defined free monads can be captured in the calculus.
- We present (in Section 5.3) a formal definition of the syntax and type system.
- We give (in Section 5.4) a categorical semantics for our calculus.
- We present (in Section 5.5) an operational semantics for our calculus, and discuss how it relates to the categorical semantics.

Section 5.6 discusses related work, and Section 5.7 concludes.

5.2 PROGRAMMING WITH EXTENSIBLE DATA TYPES, BY EXAMPLE

The basis of our calculus is the polymorphic λ-calculus extended with kinds and restricted to rank-1 polymorphism, allowing the definition of many familiar polymorphic functions, such as $(\mathsf{id} : \forall\alpha.\alpha \Rightarrow \alpha) = \lambda x.x$ or $(\mathsf{const} : \forall\alpha.\forall\beta.\alpha \Rightarrow \beta \Rightarrow \alpha) = \lambda x.\lambda y.x$. Types are closed under products and coproducts, with the unit type ($\mathbb{1}$) and empty type () acting as their respective units. Furthermore, we include a type-level fixpoint (μ), which can be used to encode many well-known algebraic data types. For example, the familiar type of lists is encoded as List $\triangleq \lambda\alpha.\mu(\lambda X.\mathbb{1} + (\alpha \times X))$. A key feature of the calculus is that all higher-order types (i.e., that have one or more type argument) are, by construction, functorial in all their arguments. While this imposes some restrictions on the types we can define, it also means that the programmer gets access to primitive mapping and folding operations that they would otherwise have to define themselves. For the type List, for example, this means that we get both the usual mapping operation transforming its elements, as well as an operation corresponding to Haskell's foldr, for free.

Although the mapping and folding primitives for first-order type constructors (i.e., those taking arguments of kind $\star$ and producing a type of kind $\star$) are already enough to solve the expression problem for regular algebraic data types (Section 5.2.2) and to encode modular algebraic effects (Section 5.2.3), they can readily be generalized to higher-order type constructors. That is, type constructors that construct higher-order types from higher-order types. The benefit of this generalization is that our calculus can also capture the definition of so-called *nested data types* [Bird and Meertens, 1998], which arise as the fixpoint of a *higher-order functor*. We make essential use of the calculus' higher-order capabilities in Section 5.2.4 to define modular handlers for scoped effects [Yang et al., 2022] and modular elaborations for higher-order effects [Bach Poulsen and Van der Rest, 2023], as in both cases effect trees that represents monadic programs with higher-order operations is defined as a nested data type.

5.2.1 *Notation*

All code examples in this section directly correspond to programs in our calculus, but we take some notational liberty to simplify the exposition. Abstraction and application of type variables is left implicit. Similarly, we omit first-order universal quantifications. By convention, we denote type variables bound by type-level λ-abstraction using capital letters (e.g., X), and those bound by universal quantification using Greek letters (e.g., α,β).

5.2.2 *Modular Interpreters in the style of Data Types à la Carte*

We consider how to define a modular interpreter for a small expression language of simple arithmetic operations. For starters, we just include literals and addition. The corresponding BNF equation and signature functor are given below:

$$e ::= \mathbb{N} \mid e + e \qquad\qquad \mathrm{Expr} \triangleq \lambda X.\mathbb{N} + (X \times X)$$

Now, we can define an eval that maps expressions—given by the fixpoint of Expr—to their result:

$$\begin{aligned}&\mathrm{expr} : \mathbb{N} + (\mathbb{N} \times \mathbb{N}) \Rightarrow \mathbb{N}\\&\mathrm{expr} = (\lambda x.x) \blacktriangledown (\lambda x.\pi_1\ x + \pi_2\ x)\end{aligned}$$

$$\begin{aligned}&\mathrm{eval} : \mu(\mathrm{Expr}) \Rightarrow \mathbb{N}\\&\mathrm{eval} = (\!|\ \mathrm{expr}\ |\!)^{\mathrm{Expr}}\end{aligned}$$

Terms typeset in **purple** are built-in operations. π_1 and π_2 are the usual projection functions for products, and $- \blacktriangledown -$ is an eliminator for coproducts. Following Meijer et al. [1991], we write $(\!|\ \mathrm{alg}\ |\!)^{\tau}$ (i.e., "banana brackets") to denote a fold over the type $\mu(\tau)$ with an *algebra* of type $\mathrm{alg} : \tau\ \tau' \Rightarrow \tau'$. The calculus does not include a general term level fixpoint; the only way to write a function that recurses on the substructures of a μ-type is by using the built-in folding operation. While this limits the operations we can define for a given type, it also ensures that all well-typed terms in the calculus have a well-defined semantics.

Now, we can extend this expression language with support for a multiplication operation as follows, where $\mathrm{Mul} \triangleq \lambda X.X \times X$:

$$\begin{aligned}&\mathrm{mul} : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N} &\qquad& \mathrm{eval} : \mu(\mathrm{Expr} + \mathrm{Mul}) \Rightarrow \mathbb{N}\\&\mathrm{mul} = \lambda x.\pi_1\ x * \pi_2\ x &\qquad& \mathrm{eval} = (\!|\ \mathrm{expr} \blacktriangledown \mathrm{mul}\ |\!)^{\mathrm{Expr} + \mathrm{Mul}}\end{aligned}$$

5.2.3 *Modular Algebraic Effects using the Free Monad*

As our second example we consider how to define modular algebraic effects and handlers [Plotkin and Pretnar, 2009a] in terms of the free monad following Swierstra [2008]. First, we define the Free type which constructs a free monad for a given signature functor f. We can think of a term with type Free $f\ \alpha$ as a syntactic representation of a monadic program producing a value of type α with f describing the operations which we can use to interact with the monadic context.

$$\mathrm{Free} : (\star \rightsquigarrow \star) \rightsquigarrow \star \rightsquigarrow \star \quad \triangleq \quad \lambda f.\lambda\alpha.\mu(\lambda X.\alpha + fX)$$

Note that the type Free is actually a functor in both its arguments, and thus there are two ways to "map over" a value of type Free $f\ \alpha$; we can transform the values at the leaves using a function $\alpha \Rightarrow \beta$, or the shape of the nodes using a natural transformation $\forall\alpha.f\ \alpha \Rightarrow g\ \alpha$. The higher-order map can be used, for example, for defining function that reorders the operations of effect trees with a composite signature.

$$\begin{aligned}&\mathrm{reorder} : \mathrm{Free}\ (f+g)\ \alpha \Rightarrow \mathrm{Free}\ (g+f)\ \alpha\\ &\mathrm{reorder} = \mathbf{map}\langle \iota_2 \blacktriangledown \iota_1 \rangle^{\mathrm{Free}}\end{aligned}$$

Here, we use higher-order instances at kind $\star \rightsquigarrow \star$ of the coproduct eliminator $- \blacktriangledown -$, the coproduct injection functions ι_1, ι_2, and the functorial map operation $\mathbf{map}\langle - \rangle^{-}$.

Effect handlers can straightforwardly be implemented as folds over Free. In fact, the behavior of a handler is entirely defined by the algebra that we use to fold over the effect tree, allowing us write a generic handle function:

$$\begin{aligned}\mathrm{handle} :\ & (\alpha \Rightarrow \beta) \Rightarrow (f\ (\mathrm{Free}\ g\ \beta) \Rightarrow \mathrm{Free}\ g\ \beta)\\ & \Rightarrow \mathrm{Free}\ (f+g)\ \alpha \Rightarrow \mathrm{Free}\ g\ \beta\\ \mathrm{handle} = {} & \lambda h.\lambda i.(\!|\ (\mathbf{in} \circ \iota_1 \circ h) \blacktriangledown i \blacktriangledown (\mathbf{in} \circ \iota_2)\ |\!)^{\alpha+(fX)+(gX)}\end{aligned}$$

Here, $\mathbf{in}$ is the constructor of a type-level fixpoint (μ). The fold above distinguishes three cases: (1) pure values, in which case we return it again using the function h; (2) an operation of the signature f which is handled using the function i; or (3) an operation of the signature g which is preserved by reconstructing the effect tree and doing nothing.

As an example, we consider how to implement a handler for the Abort effect, which has a single operation indicating abrupt termination of a computation. We define its signature functor as follows:

$$\mathrm{Abort} : \star \rightsquigarrow \star \quad \triangleq \quad \lambda X.\mathbb{1}$$

The definition of Abort ignores its argument, X, which is the type of the continuation. After aborting a computation, there is no continuation, thus the Abort effect does not need to store one. A handler for Abort is then defined like so, invoking the generic handle function defined above:

$$\begin{aligned}&\mathrm{hAbort} : \mathrm{Free}\ (\mathrm{Abort} + f)\ \alpha \Rightarrow \mathrm{Free}\ f\ (\mathrm{Maybe}\ \alpha)\\&\mathrm{hAbort} = \mathrm{handle}\ \mathrm{Just}\ (\lambda x.\mathbf{in}\ (\iota_1\ \mathrm{Nothing}))\end{aligned}$$

5.2.4 *Modular Higher-Order Effects*

To describe the syntax of computations that interact with their monadic context through higher-order operations (that is, operations whose arguments can themselves also be monadic computations) we need to generalize the free monad as follows.

$$\mathrm{Prog} : ((\star \rightsquigarrow \star) \rightsquigarrow \star \rightsquigarrow \star) \rightsquigarrow \star \rightsquigarrow \star \triangleq \lambda f.\mu(\lambda X.\lambda \alpha.\alpha + (f\ X\ \alpha))$$

Note that, unlike the Free type, Prog is defined as the fixpoint of a higher-order functor. This generalization allows for signature functors to freely choose the return type of continuations. Following Yang et al. [2022], we use this additional expressivity to describe the syntax of higher-order operations by nesting continuations. For example, the following defines the syntax of an effect for exception catching, that we can interact with by either throwing an exception, or by declaring an exception handler that first executes its first argument, and only runs the second computation if an exception was thrown.

$$\mathrm{Catch} : (\star \rightsquigarrow \star) \rightsquigarrow \star \rightsquigarrow \star \quad \triangleq \quad \lambda X.\lambda \alpha.\mathbb{1} + (X(X\alpha) \times (X(X\alpha))$$

A value of type Prog Catch α is then a syntactic representation of a monadic program that can both throw and catch exceptions. From this syntactic representation we can proceed in two different ways. The first option is to replace exception catching with an application of the hAbort handler, in line with Plotkin and Pretnar [2009a] original strategy for capturing higher-order operations. In recent work, Bach Poulsen and Van der Rest [2023] demonstrated how such abbreviations can be made modular and reusable by implementing them as algebras over the Prog type. Following their approach, we define the following elaboration of exception catching into a first-order effect tree.

$$
\begin{aligned}
&\mathrm{eCatch} : \mathrm{Prog\ Catch}\ \alpha \Rightarrow \mathrm{Free\ Abort}\ \alpha\\
&\mathrm{eCatch} = (\!|\ (\mathbf{in} \circ \iota_1)\\
&\qquad\qquad \blacktriangledown\ (\mathbf{in} \circ \iota_2)\\
&\qquad\qquad \blacktriangledown\ (\lambda x.\mathrm{hAbort}\ (\pi_1\ x) \gg\!= \mathrm{maybe}\ (\mathrm{join}\ (\pi_2\ x))\ \mathrm{id})\\
&\qquad\qquad |\!)^{\alpha + \mathrm{Catch}\ X\ \alpha}
\end{aligned}
$$

Here, the applications of monadic bind ($\gg\!=$) and join refer to the monadic structure of Free. Alternatively, we can define a handler for exception catching directly by folding over the Prog type, following the *scoped effects* approach by Wu et al. [2014]:

$$
\begin{aligned}
&\mathrm{hCatch} : \mathrm{Prog}\ (\mathrm{Catch} + h)\ \alpha \Rightarrow \mathrm{Prog}\ h\ (\mathrm{Maybe}\ \alpha)\\
&\mathrm{hCatch} = (\!|\ (\mathbf{in} \circ \iota_1 \circ \mathrm{Just})\\
&\qquad\qquad \blacktriangledown\ (\lambda x.\mathbf{in}\ (\iota_1\ \mathrm{Nothing}))\\
&\qquad\qquad \blacktriangledown\ (\lambda x.\pi_1\ x \gg\!= \mathrm{maybe}\ (\pi_2\ x \gg\!= \mathrm{fwd})\ \mathrm{id}))\\
&\qquad\qquad \blacktriangledown\ (\mathbf{in} \circ \iota_2)\ |\!)^{\alpha + (\mathrm{Catch}\ X\ \alpha) + (h\ X\ \alpha)}
\end{aligned}
$$

Where the function fwd establishes that Maybe commutes with the Prog type in a suitable way:

$$\mathrm{fwd} : \mathrm{Maybe}\ (\mathrm{Prog}\ h\ (\mathrm{Maybe}\ \alpha)) \Rightarrow \mathrm{Prog}\ h\ (\mathrm{Maybe}\ \alpha)$$

That is, we show that Prog h is a *modular carrier* for Maybe [Schrijvers et al., 2019].

As demonstrated, our calculus supports defining higher-order effects and their interpretations. To conveniently sequence higher-order computations we typically also want to use a monadic bind function, such as $\gg\!= : \mathrm{Prog}\ h\ \alpha \to (\alpha \to \mathrm{Prog}\ h\ \beta) \to \mathrm{Prog}\ h\ \beta$. While it is possible to define monadic bind for *Free* from Section 5.2.3 in terms of a plain fold, defining the monadic bind

$$
\begin{array}{rclcl}
 & & \alpha, \beta, \gamma, X, Y & \in & \text{String} \\
\\
\mathit{Kind} & \ni & k & ::= & \star \mid k \rightsquigarrow k \\
\mathit{KindEnv} & \ni & \Delta, \Phi & ::= & \emptyset \mid \Delta, \alpha : k \\
\\
\mathit{Type} & \ni & \tau & ::= & \alpha \mid X \mid \tau\ \tau \mid \lambda X.\tau \mid \mu(\tau) \mid \tau \Rightarrow \tau \\
 & & & \mid & \mid \mathbb{1} \mid \tau \times \tau \mid \tau + \tau \\
\mathit{Scheme} & \ni & \sigma & ::= & \forall \alpha.\sigma \mid \tau
\end{array}
$$

Figure 16: Type syntax

for *Prog* generally requires a *generalized fold* [Bird and Paterson, 1999, Yang et al., 2022]. Adding this and other recursion principles [Meijer et al., 1991] to our calculus is future work.

5.3 THE CALCULUS

The previous section demonstrated how a language with built-in support for functors, folds, and fixpoints provides support for defining and working with state-of-the-art techniques for type safe modular programming. In this section we present a core calculus for such a language. The basis of our calculus is the first-order fragment of System F^{ω}—i.e., the polymorphic λ-calculus with kinds, where universal quantification is limited to prenex normal form à la Hindley-Milner. Additionally, the syntax of types, defined in Figure 16, includes primitives for constructing recursive types ($\mu(-)$), products ($\times$) and coproducts ($+$), as well as a unit type ($\mathbb{1}$) and empty type (). In the definition of the syntax of types, the use of $\forall$-types is restricted by stratifying the syntax into two layers, types and type schemes. Consequently, our calculus is, by design, *predicative*: $\forall$-types can quantify over types but not type schemes.

The motivation for this predicative design is that it permits a relatively straightforward categorical interpretation of $\forall$-types in terms of *ends* (see Section 5.4.2.3). Whereas the restriction of universal quantification to prenex normal form is usually imposed to facilitate type inference, our calculus does not support inference in its current form due to the structural treatment of data types. In a structural setting, inference requires the reconstruction of (recursive) data type definitions from values, which is, in general, not possible.

We remark that the current presentation of the type system is *declarative*, meaning certain algorithmic aspects crucial for type checking, such as normalization and equality checking of types, are not covered in the current exposition. Regarding decidability of the type system: our system is a subset of System F_ω, whose Church-style formulation is decidable while its Curry-style formulation is not. As such, we expect our type system to inherit these properties. Since we are restricting ourselves to a predicative subset of F_ω, we are optimistic that the Curry-style formulation of our type system will be decidable too, but verifying this expectation is future work.

5.3.1 *Well-Formed Types*

Types are well-formed with respect to a kind k, describing the arity of a type's parameters, if it has any. Well-formedness of types is defined using the judgment $\Delta \mid \Phi \vdash \tau : k$, stating that the type τ has kind k under contexts Δ and Φ. Similarly, well-formedness of type schemes is defined by the judgment $\Delta \vdash \sigma$, stating that the type scheme σ is well-formed with respect to the context Δ.

Following Johann et al. [2021], well-formedness of types is defined with respect to two contexts, one containing functorial variables (Φ), and one containing variables with mixed variance (Δ). Specifically, the variables in the context Φ are restricted to occur only in *strictly positive* [Abbott et al., 2005b, Coquand and Paulin, 1988] positions (i.e., they can never appear to the left of a function arrow), while the variables in Δ can have mixed variance. This restriction on the occurrence of the variables in Φ is enforced in the well-formedness rule for function types, K-Fun, which requires that its domain is typed under an empty context of functorial variables, preventing the domain type from derefer-

$$\boxed{\Delta \mid \Phi \vdash \tau : k}$$

$$\textsc{K-Var}\ \frac{k : \alpha \in \Delta}{\Delta \mid \Phi \vdash \alpha : k} \qquad \textsc{K-Fvar}\ \frac{\Phi(X) \mapsto k}{\Delta \mid \Phi \vdash X : k} \qquad \textsc{K-App}\ \frac{\Delta \mid \Phi \vdash \tau_1 : k_1 \rightsquigarrow k_2 \qquad \Delta \mid \Phi \vdash \tau_2 : k_1}{\Delta \mid \Phi \vdash \tau_1\ \tau_2 : k_2}$$

$$\textsc{K-Abs}\ \frac{\Delta \mid \Phi, (X \mapsto k_1) \vdash \tau : k_2}{\Delta \mid \Phi \vdash \lambda X.\tau : k_1 \rightsquigarrow k_2} \qquad \textsc{K-Fix}\ \frac{\Delta \mid \Phi \vdash \tau : k \rightsquigarrow k}{\Delta \mid \Phi \vdash \mu(\tau) : k} \qquad \textsc{K-Fun}\ \frac{\Delta \mid \emptyset \vdash \tau_1 : \star \qquad \Delta \mid \Phi \vdash \tau_2 : \star}{\Delta \mid \Phi \vdash \tau_1 \Rightarrow \tau_2 : \star}$$

$$\textsc{K-Empty}\ \frac{}{\Delta \mid \Phi \vdash \ : k} \qquad \textsc{K-Unit}\ \frac{}{\Delta \mid \Phi \vdash \mathbb{1} : k} \qquad \textsc{K-Product}\ \frac{\Delta \mid \Phi \vdash \tau_1 : k \qquad \Delta \mid \Phi \vdash \tau_2 : k}{\Delta \mid \Phi \vdash \tau_1 \times \tau_2 : k}$$

$$\textsc{K-Sum}\ \frac{\Delta \mid \Phi \vdash \tau_1 : k \qquad \Delta \mid \Phi \vdash \tau_2 : k}{\Delta \mid \Phi \vdash \tau_1 + \tau_2 : k}$$

$$\boxed{\Delta \vdash \sigma}$$

$$\textsc{SC-Forall}\ \frac{\Delta, (\alpha \mapsto k) \vdash \sigma}{\Delta \vdash \forall \alpha.\sigma} \qquad \textsc{SC-Type}\ \frac{\Delta \mid \emptyset \vdash \tau : \star}{\Delta \vdash \tau}$$

Figure 17: Well-formedness rules for types and type schemes

encing any functorial variables bound in the surrounding context. While it may seem overly restrictive to require type expressions to be strictly positive—rather than merely positive—in Φ, this is necessary to ensure that μ-types, as well as its introduction and elimination forms, have a well-defined semantics (see Section 5.4.2.1). Variables in Φ are bound by type-level λ-abstraction, meaning that any type former with kind $k_1 \rightsquigarrow k_2$ is functorial in its argument. In contrast, the variables in Δ are bound by $\forall$-quantification.

Products ($\times$), coproducts ($+$), units ($\mathbb{1}$) and empty types () can be constructed at any kind, reflecting the fact that the corresponding categorical (co)limits can be lifted from Set to its functor categories by computing them pointwise. This pointwise lifting of these (co)limits to functor categories is reflected in the

β equalities for these type formers (shown in Figure 20), which allow an instance at kind $k_1 \rightsquigarrow k_2$, when applied with a type argument, to be replaced with an instance at kind k_2.

The well-formed judgements for types effectively define a (simply typed) type level λ-calculus with base "type" $\star$. Consequently, the same type has multiple equivalent representations in the presence of β-redexes, raising the question of how we should deal with type normalization. The approach we adopt here is to add a non-syntactic conversion rule to the definition of our type system that permits any well-formed term to be typed under an equivalent type scheme. Section 5.3.3 discusses type equivalence in more detail.

5.3.2 *Well-Typed Terms*

Figure 18 shows the term syntax of our calculus. Along with the standard syntactic forms of the polymorphic λ-calculus we include explicit type abstraction and application, as well as introduction and elimination forms for recursive types (**in**/**unin**), products ($\pi_1/\pi_2/- \blacktriangle -$), coproducts ($\iota_1/\iota_2/- \blacktriangledown -$), and the unit (**tt**) and empty (**absurd**) types. Furthermore, the calculus includes dedicated primitives for mapping ($\mathbf{map}\langle - \rangle^{-}$) and folding ($(\!| - |\!)^{-}$) over a type.

Figure 18 also includes the definition of *arrow types*. In spirit of the syntactic notion of natural transformations used by Abel and Matthes [2002], Abel et al. [2003, 2005] to study generalized (Mendler) iteration, an arrow type of the form $\tau_1 \xrightarrow{k} \tau_2$ (where $\tau_1, \tau_2 : k$) defines the type of *morphisms* between the objects that interpret τ_1 and τ_2. Arrow types are defined by induction over k, since the precise meaning of morphism for any pair of types depends on their kind. If $k = \star$, then a morphism between τ_1 and τ_2 is simply a function type. However, if τ_1 and τ_2 have one or more type argument, they are to be interpreted as objects in a suitable functor category, meaning that their morphisms are natural transformations. This is reflected in the definition of arrow types, by unfolding an arrow $\tau_1 \xrightarrow{k} \tau_2$ to a $\forall$-type that closes over all type arguments of τ_1 and τ_2, capturing the intuition that

$$x, y \in \text{String}$$

$$\begin{array}{lclcl}
\textit{Env} & \ni & \Gamma & ::= & \emptyset \mid \Gamma, x : \sigma \\
\textit{Term} & \ni & M, N & ::= & x \mid M\ N \mid \lambda x.M \mid \mathbf{let}\ (x : \sigma) = M\ \mathbf{in}\ N \\
 & & & \mid & \Lambda\alpha.M \mid M\ @\tau \mid \mathbf{in} \mid \mathbf{unin} \mid \mathbf{map}\langle M \rangle^{\tau} \mid (\!| M |\!)^{\tau} \\
 & & & \mid & \pi_1 \mid \pi_2 \mid M \blacktriangle N \mid \iota_1 \mid \iota_2 \mid M \blacktriangledown N \mid \mathbf{tt} \mid \mathbf{absurd}
\end{array}$$

$$\begin{array}{rcll}
\tau_1 \xrightarrow{\star} \tau_2 & \triangleq & \tau_1 \Rightarrow \tau_2 & \text{(Arrow Types)} \\
\tau_1 \xrightarrow{(k_1 \rightsquigarrow k_2)} \tau_2 & \triangleq & \forall \alpha.\ \tau_1\ \alpha \xrightarrow{k_2} \tau_2\ \alpha & \\
\textit{where} & & \Delta \vdash \tau_1 \xrightarrow{k} \tau_2 \quad \textit{if} \quad \Delta \mid \emptyset \vdash \tau_1, \tau_2 : k &
\end{array}$$

Figure 18: Term syntax

polymorphic functions cor respond to natural transformations.[55] For instance, we would type the inorder traversal of binary trees as *inorder* : *Tree* $\xrightarrow{\star \rightsquigarrow \star}$ *List* ($\triangleq \forall \alpha.\text{Tree}\ \alpha \Rightarrow \text{List}\ \alpha$), describing a natural transformation between the *Tree* and *List* functors.

[55] *This intuition is made formal by Theorem 1 in Section 5.4.4.*

The typing rules are shown in shown in Figure 19. The rules rely on arrow types for introduction and elimination forms. For example, Products can be constructed at any kind (following rule K-Product in Figure 17), so the rules for terms that operate on these (i.e., T-Fst, T-Snd, and T-Fork) use arrow types at any kind k. Consequently, arrow types should correspond to morphisms in a suitable category, such that the semantics of a product type and its introduction/elimination forms can be expressed as morphisms in this category.

5.3.3 *Type Equivalence*

In the presence of type level λ-abstraction and application, the same type can have multiple representations. For this reason, the type system defined in Figure 19 includes a non-syntactic conversion rule that allows a well-typed term to be re-typed under any equivalent type scheme. The relevant equational theory for types is defined in Figure 20, and includes the customary β and η equivalences for λ-terms, as well as β rules for product,

$$\boxed{\Gamma \vdash M : \sigma}$$

$$\textsc{T-Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textsc{T-App}\ \frac{\Gamma \vdash M : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2} \qquad \textsc{T-Abs}\ \frac{\Gamma, (x : \tau_1) \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \Rightarrow \tau_2}$$

$$\textsc{T-Let}\ \frac{\Gamma \vdash M : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash N : \sigma_2}{\Gamma \vdash \mathbf{let}\ (x : \sigma_1) = M\ \mathbf{in}\ N : \sigma_2} \qquad \textsc{T-TypeAbs}\ \frac{\Gamma \vdash M : \sigma \qquad \alpha \notin \mathsf{freevars}(\Gamma)}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma} \qquad \textsc{T-TypeApp}\ \frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M\ @\tau : \sigma[\tau/\alpha]}$$

$$\textsc{T-In}\ \frac{}{\Gamma \vdash \mathbf{in} : \tau\ \mu(\tau) \xrightarrow{k} \mu(\tau)} \qquad \textsc{T-Out}\ \frac{}{\Gamma \vdash \mathbf{unin} : \mu(\tau) \xrightarrow{k} \tau\ \mu(\tau)} \qquad \textsc{T-Map}\ \frac{\Gamma \vdash M : \tau_1 \xrightarrow{k_1} \tau_2}{\Gamma \vdash \mathbf{map}\langle M \rangle^{\tau} : \tau\ \tau_1 \xrightarrow{k_2} \tau\ \tau_2}$$

$$\textsc{T-Fold}\ \frac{\Gamma \vdash M : \tau_1\ \tau_2 \xrightarrow{k} \tau_2}{\Gamma \vdash (\!| M |\!)^{\tau_1} : \mu(\tau_1) \xrightarrow{k} \tau_2} \qquad \textsc{T-Fst}\ \frac{}{\Gamma \vdash \pi_1 : \tau_1 \times \tau_2 \xrightarrow{k} \tau_1} \qquad \textsc{T-Snd}\ \frac{}{\Gamma \vdash \pi_2 : \tau_1 \times \tau_2 \xrightarrow{k} \tau_2}$$

$$\textsc{T-Fork}\ \frac{\Gamma \vdash M : \tau \xrightarrow{k} \tau_1 \qquad \Gamma \vdash N : \tau \xrightarrow{k} \tau_2}{\Gamma \vdash M \blacktriangle N : \tau \xrightarrow{k} \tau_1 \times \tau_2} \qquad \textsc{T-Inl}\ \frac{}{\Gamma \vdash \iota_1 : \tau_1 \xrightarrow{k} \tau_1 + \tau_2}$$

$$\textsc{T-Inr}\ \frac{}{\Gamma \vdash \iota_2 : \tau_2 \xrightarrow{k} \tau_1 + \tau_2} \qquad \textsc{T-Join}\ \frac{\Gamma \vdash M : \tau_1 \xrightarrow{k} \tau \qquad \Gamma \vdash M : \tau_2 \xrightarrow{k} \tau}{\Gamma \vdash M \blacktriangledown N : \tau_1 + \tau_2 \xrightarrow{k} \tau} \qquad \textsc{T-Unit}\ \frac{}{\Gamma \vdash \mathbf{tt} : \mathbb{1}}$$

$$\textsc{T-Empty}\ \frac{}{\Gamma \vdash \mathbf{absurd} : \xrightarrow{k} \tau} \qquad \textsc{T-Conv}\ \frac{\Gamma \vdash M : \sigma_1 \qquad \sigma_1 \equiv \sigma_2}{\Gamma \vdash M : \sigma_2}$$

Figure 19: Well-formed terms

sum, unit, and empty types. The equations shown in Figure 20 are motivated by the semantic model we discuss in Section 5.4, in the sense that equivalent types are interpreted to naturally isomorphic functors. The relation is also reflexive and transitive, motivated by respectively the identity and composition of natural isomorphisms. Viewing the equalities in Figure 20 left-to-right

$$\begin{aligned}
(\lambda X.\tau_1)\ \tau_2 &\equiv \tau_1[\tau_2/X] \\
(\lambda X.\tau\ X) &\equiv \tau \\
(\tau_1 \times \tau_2)\ \tau &\equiv (\tau_1\ \tau) \times (\tau_2\ \tau) \\
(\tau_1 + \tau_2)\ \tau &\equiv (\tau_1\ \tau) + (\tau_2\ \tau) \\
\mathbb{1}\ \tau &\equiv \mathbb{1} \\
\tau &\equiv
\end{aligned}$$

$$\begin{aligned}
\mathsf{T} \;:=\;& [\,] \mid \mathsf{T}\ \tau \mid \tau\ \mathsf{T} \mid \mu(\mathsf{T}) \mid \mathsf{T} \Rightarrow \tau \mid \tau \Rightarrow \mathsf{T} \mid \\
& \mathsf{T} \times \tau \mid \tau \times \mathsf{T} \mid \mathsf{T} + \tau \mid \tau + \mathsf{T}
\end{aligned}$$

$$\frac{\tau_1 \equiv \tau_2}{\mathsf{T}[\ \tau_1\ ] \equiv \mathsf{T}[\ \tau_2\ ]}$$

Figure 20: Equational theory for types

provides us with a basis for a normalization strategy for types, which would be required for implementing the type system.

5.4 CATEGORICAL SEMANTICS

In this section, we consider how to define a categorical semantics for our calculus, drawing inspiration from the semantics defined by Johann and Polonsky [2019] and Johann et al. [2021], Johann and Ghiorzi [2021]. To define this semantics, we must show that each type in our calculus corresponds to a functor, and that all such functors have initial algebras. In Section 5.4.3 we discuss the requirements for these initial algebras to exist, and argue informally why they should exist for the functors interpreting our types. Although Johann and Polonsky [2019] present a detailed argument for the existence of initial algebras of the functors underlying nested data types, it is still future work to adapt this argument to our setting.

The general setup of our semantics is to interpret types of kind $\star$ as objects in Set (the category of sets), higher-order types as functors on Set, and type schemes as objects in Set_2 (the category of large sets). This size bump is necessary to model the universal quantification over types in type schemes. Crucially, Set is a *full subcategory* of Set_1, as witnessed by the existence of a fully faithful inclusion functor I:

$$\text{Set} \overset{I}{\hookrightarrow} \text{Set}_1$$

Assuming cumulative universes (i.e., the collection of all large sets also includes all small sets), I is just the identity functor. We remark that both $\textsc{Set}$ and $\textsc{Set}_1$ are *complete* and *cocomplete* and *cartesian closed*. Importantly, since I is fully faithful, the cartesian closed structure of $\textsc{Set}$ is reflected in $\textsc{Set}_1$ for those objects that lie in the image of I.

The subcategory relation between $\textsc{Set}$ and $\textsc{Set}_1$ reflects the syntactic restriction of types to rank-1 polymorphism: all objects in $\textsc{Set}$ can also be found in $\textsc{Set}_1$, but $\textsc{Set}_1$ is sufficiently larger than $\textsc{Set}$ that it also includes objects modelling quantification over objects in $\textsc{Set}$. This intuition is embodied by fact that every functor $F : \mathcal{C}^{op} \times \mathcal{C} \to \textsc{Set}_1$, where $\mathcal{C}$ is smaller than $\textsc{Set}_1$ (which includes $\textsc{Set}$), has an end in $\textsc{Set}_1$. This follows from completeness of $\textsc{Set}_1$ [MacLane, 1971, p. 224, corollary 2]. We discuss the use of ends for modelling universal quantification in more detail in Section 5.4.2.3.

5.4.1 *Interpreting Kinds and Kind Environments*

We associate with each kind k a category whose objects interpret the types of that kind. The semantics of kinds is defined by induction over k, where we map the base kind $\star$ to $\textsc{Set}$, and kinds of the form $k_1 \rightsquigarrow k_2$ to the category of functors between their domain and codomain.[56]

[56] Here, **CAT** denotes the (very large) category of large categories. Although $\textsc{Set}$ itself is locally small, its functor categories have a large set of morphisms.

$$\begin{aligned} [\![-]\!] &: \mathit{Kind} \to \mathbf{CAT} \\ [\![\star]\!] &= \textsc{Set} \\ [\![k_1 \rightsquigarrow k_2]\!] &= [\, [\![k_1]\!] \,,\, [\![k_2]\!] \,] \end{aligned}$$

By interpreting types of kind $k_1 \rightsquigarrow k_2$ as objects in a functor category, we formalize the intuition that higher-order types correspond to functors. The semantics of kind contexts is then defined on a per-entry basis, as a chain of products of the categories that interpret their elements.

$$\begin{aligned} [\![-]\!] &: \mathit{Context} \to \mathbf{CAT} \\ [\![\emptyset]\!] &= \bullet \\ [\![\Delta, \alpha \mapsto k]\!] &= [\![\Delta]\!] \times [\![k]\!] \end{aligned}$$

Here, $\bullet$ denotes the *trivial category*, which has a single object, $*$, together with its identity morphism, id_*. It is worth mentioning that $\bullet$ and $- \times -$, together with the operation of constructing a functor category, $[-,-]$, imply that **CAT** is a cartesian closed category. We will use this cartesian closed structure to give a semantics to the fragment of well-formed types that corresponds to the simply-typed λ-calculus.

5.4.2 *Interpreting Types*

Since a well-formed type $\Delta \mid \Phi \vdash \tau : k$ is intended to be functorial in all variables in Φ, it is clear that its semantics should be a functor over the category associated with Φ (i.e., $[\![\Phi]\!]$). But what about the variables in Δ, which can occur both in covariant and contravariant positions? For example, in the type of the identity function, $\forall\alpha.\alpha \Rightarrow \alpha$, we cannot interpret the sub-derivation for $\alpha \Rightarrow \alpha$ as a functor over the category interpreting its free variables since there would not be a sensible way to define its action on morphisms due to the negative occurence of α. To account for the mixed variance of universally quantified type variables, we instead adopt a *difunctorial semantics*, interpreting types as a functor on the product category $[\![\Delta]\!]^{op} \times [\![\Delta]\!]$ (similar representations of type expressions with mixed variance appear, for example, when considering Mendler-style inductive types [Uustalu and Vene, 1999], or the object calculus semantics by Glimming and Ghani [2004]). Well-formed types (left) and type schemes (right) are interpreted as a functors over their contexts of the following form:

$$[\![\, \Delta \mid \Phi \vdash \tau : k \,]\!] : ([\![\Delta]\!]^{op} \times [\![\Delta]\!]) \times [\![\Phi]\!] \to [\![k]\!]$$

$$[\![\, \Delta \vdash \sigma \,]\!] : [\![\Delta]\!]^{op} \times [\![\Delta]\!] \to \textsc{Set}_1$$

Ultimately, the goal of this setup is to interpret $\forall$-types as *ends* in $\textsc{Set}_1$, which allows us to formally argue that terms that are well-formed with an arrow type of the form $\tau_1 \xrightarrow{k} \tau_2$ (which unfolds to $\forall\bar{\alpha}.\tau_1\ \bar{\alpha} \Rightarrow \tau_2\ \bar{\alpha}$) correspond, in a suitable sense, to the natural transformations between the functors interpreting τ_1 and τ_2. Or, put differently, terms with an arrow type define a morphism between the interpretation of their domain and codomain. We discuss the semantics of universal quantification

$$
\begin{aligned}
[\![\Delta \mid \Phi \vdash \alpha : \tau]\!] &= \mathbf{lookup}^{\Delta}_{\alpha} \circ \pi_2 \circ \pi_1 \\
[\![\Delta \mid \Phi \vdash X : \tau]\!] &= \mathbf{lookup}^{\Phi}_{X} \circ \pi_2 \\
[\![\Delta \mid \Phi \vdash \tau_1\ \tau_2 : k_2]\!] &= \mathsf{eval} \circ \langle\ [\![\Delta \mid \Phi \vdash \tau_1 : k_1 \rightsquigarrow k_2]\!], [\![\Delta \mid \Phi \vdash \tau_2 : k_1]\!]\ \rangle \\
[\![\Delta \mid \Phi \vdash \lambda X.\tau : k_1 \rightsquigarrow k_2]\!] &= \mathsf{curry}([\![\Delta \mid \Phi, X : k_1 \vdash \tau : k_2]\!]) \\
[\![\Delta \mid \Phi \vdash \mu(\tau) : k]\!] &= \boldsymbol{\mu}([\![\Delta \mid \Phi \vdash \tau : k \rightsquigarrow k]\!]) \\
[\![\Delta \mid \Phi \vdash \tau_1 \Rightarrow \tau_2 : \star]\!] &= \mathbf{exp}([\![\Delta \mid \emptyset \vdash \tau_1 : \star]\!], [\![\Delta \mid \Phi \vdash \tau_2 : \star]\!]) \\
[\![\Delta \mid \Phi \vdash \ : \star]\!] &= \bot \\
[\![\Delta \mid \Phi \vdash \mathbb{1} : \star]\!] &= \top \\
[\![\Delta \mid \Phi \vdash \tau_1 \times \tau_2 : k]\!] &= [\![\Delta \mid \Phi \vdash \tau_1 : k]\!] \times [\![\Delta \mid \Phi \vdash \tau_2 : k]\!] \\
[\![\Delta \mid \Phi \vdash \tau_1 + \tau_2 : k]\!] &= [\![\Delta \mid \Phi \vdash \tau_1 : k]\!] + [\![\Delta \mid \Phi \vdash \tau_2 : k]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![\Delta \vdash \forall \alpha.\sigma]\!] &= \mathbf{end}(\mathsf{curry}([\![\Delta, \alpha : k \vdash \sigma]\!] \circ \mathbf{sift})) \\
[\![\Delta \vdash \tau]\!] &= I \circ [\![\Delta \mid \emptyset \vdash \tau : \star]\!]
\end{aligned}
$$

Figure 21: Semantics of well-formed types and type schemes

further in Section 5.4.2.3, and give a more precise account of the relation between arrow types and natural transformations in Section 5.4.4.

Figure 21 defines the semantics of well-formed types and type schemes. The interpretation of the empty type, unit type, and (co)product types follow immediately from (co)completeness of Set. Since they can be constructed at any kind, the semantics of (co)product types depends crucially on the fact that functor categories preserve all (co)limits of their codomain category, which implies that $[\![k]\!]$ is (co)complete for any k. To interpret variables, we utilize the cartesian closed structure of **CAT** to compute an appropriate projection based on the position of the variable in the environment.

$$
\begin{array}{lcll}
\mathbf{lookup}^{\Delta}_{\alpha} & : & [\![\ \Delta\ ]\!] \to [\![\ k\ ]\!] & \\
\mathbf{lookup}^{\Delta,\alpha:k}_{\alpha} & \mapsto & \pi_2 & \\
\mathbf{lookup}^{\Delta,\beta:k}_{\alpha} & \mapsto & \mathbf{lookup}^{\Delta}_{\alpha} \circ \pi_1 & \quad (\text{where } \alpha \neq \beta)
\end{array}
$$

Similarly, the cartesian closed structure of **CAT** also implies the existence of functors $\mathsf{eval} : [\mathcal{C}, \mathcal{D}] \times \mathcal{C} \to \mathcal{D}$ and $\mathsf{curry}(F) : \mathcal{C} \to [\mathcal{D}, \mathcal{E}]$, for any $F : \mathcal{C} \times \mathcal{D} \to \mathcal{E}$, which immediately provide a semantics for type-level application and abstraction respectively. The remaining type and type scheme constructors are interpreted using specifically-defined functors. Although their definitions are typical examples of how (co)limits are lifted to functor categories by computing them pointwise, we discuss the definition of these functors separately and in more detail respectively in Section 5.4.2.1 (recursive types), Section 5.4.2.2 (function types), and Section 5.4.2.3 ($\forall$-types).

5.4.2.1 *Recursive Types*

Following the usual categorical interpretation of inductive data types [Goguen, 1976], the semantics of recursive types is given by an *initial algebras*. We summarize the setup here. An F-*algebra* for an endofunctor $F : \mathcal{C} \to \mathcal{C}$ is defined as a tuple (A, α) of an object $A \in C$ (called the *carrier*), and a morphism $\alpha : FA \to A$. An *algebra homomorphism* between F-algebras (A, α) and (B, β) is given by a morphism $f : A \to B$ such that the following diagram commutes.

$$\begin{tikzcd}
FA \arrow[r, "\alpha"] \arrow[d, "Ff"'] & A \arrow[d, "f"] \\
FB \arrow[r, "\beta"'] & B
\end{tikzcd}$$

F-algebras and their homomorphisms form a category. If F is an endofunctor, we denote the initial object of the category of F-algebras (which, if it exists, we refer to as the initial algebra) as $(\mu F, \mathsf{in})$. Initial algebras give a semantics to inductive data types, with their universal property providing an induction principle. Given an F-algebra (A, α), we denote unique F-algebra homomorphism that factors through A by $\mathsf{cata}(\alpha) : \mu F \to A$. Instantiating the diagram above with $\mathsf{cata}(\alpha)$ gives us the familiar universal property of folds, $\mathsf{cata}(\alpha) \circ \mathsf{in} = \alpha \circ F(\mathsf{cata}(\alpha))$, which defines their computational behavior.

To interpret recursive types in our calculus, we construct the functor $\boldsymbol{\mu}(F)$, which sends objects pointwise to the initial algebras of a functor $F : \mathcal{C} \to [\mathcal{D}, \mathcal{D}]$. For a morphism $f : X \to Y$, the action of $\mu(F)$ on f is defined by factoring through the algebra defined by precomposing the initial algebra of $F(Y)$ with the action of F on f, which defines a natural transformation $F(X) \dot{\to} F(Y)$, at component $\mu(F(Y))$.

$$\begin{array}{lcl} \mu(F)(-) & : & \mathcal{C} \to \mathcal{D} \\ \mu(F)(x) & \mapsto & \mu(F(x)) \\ \mu(F)(f) & \mapsto & \mathsf{cata}(\mathsf{in} \circ F(f)_{\mu(F(Y))}) \end{array}$$

In general, it is not guaranteed that an initial algebra exists for any endofunctor $F : \mathcal{C} \to \mathcal{C}$. Typically, the existence of an initial algebras is shown by iterating F and showing that it converges, applying the classic theorem by Adámek [1974]. This approach imposes some additional requirements on the functor F and underlying category $\mathcal{C}$, which we discuss in more detail in Section 5.4.3.

5.4.2.2 *Function Types*

The functor $\mathbf{exp}(-)$ is defined by mapping onto exponential objects in SET. But we have to take some additional care to ensure that we can still define its action on morphism, as the polarity of free variables is reversed in domain of a function type. Indeed, when computed pointwise, exponential objects give rise to a bifunctor of the form $\mathcal{C}^{op} \to \mathcal{C} \to \mathcal{C}$, meaning that functors are not, in general, closed under exponentiation. To some extent we anticipated this situation already in the design of our type system by defining the well-formedness rule for function types such that the context of functorial variables, Φ, is discarded in its domain. Of course, the variables in Δ can occur both in covariant and contravariant positions, but by adopting a difunctorial semantics we limit ourselves to a specific class of functors that is closed under exponentiation. The key observation is that constructing the opposite category of the product of a category and its opposite is an idempotent (up to isomorphism) operation. That is, we have the following equivalence of categories: $(C^{op} \times \mathcal{C})^{op} \simeq C^{op} \times \mathcal{C}$. As a result, a pointwise mapping of difunctors to exponential objects does give rise to a new difunctor. We use this fact to our advantage to define the following functor $\mathbf{exp}(F, G)$ for functors $F : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{E}$ and $G : (C^{op} \times \mathcal{C}) \times \mathcal{D} \to \mathcal{E}$, of which the interpretation of function types is an instance.

$$\begin{aligned}
&\mathbf{exp}(F, G)(-) && : && (\mathcal{C}^{op} \times \mathcal{C}) \times \mathcal{D} \to \mathcal{E} \\
&\mathbf{exp}(F, G)((x, y), z) && \mapsto && G((x, y), z)^{F(y,x)} \\
&\mathbf{exp}(F, G)((f, g), h) \\
&\quad \mapsto \mathsf{curry}(G((f, g), h) \circ \mathsf{eval} \circ (id_{\mathbf{exp}(F,G)((x,y),z)} \times F(g, f))
\end{aligned}$$

We remark that $\mathbf{exp}(F,G)$ does not define an exponential object in the functor category $[(\mathcal{C}^{op} \times \mathcal{C}) \times \mathcal{D}, \mathcal{E}]$. Fortunately, for defining the semantics of term level λ-abstraction or application it is sufficient that the action on objects maps to exponentials in Set.

5.4.2.3 *Universal quantification*

The semantics of universal quantifications is expressed in terms of ends in the category Set_1. If $F : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{D}$ is a functor, then an *end* of F is an object $\int_{x \in \mathcal{C}} F(x, x) \in \mathcal{D}$ equipped with a projection map given by an extranatural transformation $\pi_x : \int_{c \in \mathcal{C}} F(c, c) \to F(x, x)$. Formally, and end of the F is defined as the universal wedge of the following diagram:

$$F(x, x) \xrightarrow{F(id_x,f)} F(x, y) \xleftarrow{F(f,id_y)} F(y, y)$$

For all $x, y \in \mathcal{C}$ and $f : x \to y$. The universal property of ends then states that any other wedge $W \in \mathcal{D}$ with maps $i : W \to F(x, x)$ and $j : W \to F(y, y)$ uniquely factors through $\int_{c \in \mathcal{C}} F(c, c)$.

$$\begin{tikzcd}
W \arrow[r, "i"] \arrow[dr, "j" near start] \arrow[d, dotted, "\mathsf{factor}(W)"'] & F(x, x) \arrow[dr, "F(id_x,f)"] & \\
\int_{c \in \mathcal{C}} F(c, c) \arrow[ur, "\pi_x"' near end] \arrow[r, "\pi_y"'] & F(y, y) \arrow[r, "F(f,id_y)"'] & F(x, y)
\end{tikzcd}$$

To model the more general situation where a $\forall$-quantified type can contain free variables that are bound by another quantifier above it in the lexical hierarchy, we define the semantics of universal quantification in terms of the *end functor*, $\mathbf{end}(-)$, which for a functor $G : \mathcal{C} \to [\mathcal{D}^{op} \times \mathcal{D}, \mathcal{E}]$ defines a functor $\mathbf{end}(G) : \mathcal{C} \to E$

whose object action is computed pointwise from ends in $\mathcal{E}$. Its action on morphisms, $\mathbf{end}(f) : \int_{d \in \mathcal{D}} G(X)(d,d) \to \int_{d \in \mathcal{D}} G(Y)(d,d)$, follows from the universal property stated above. To define the action on morphisms, we observe that the object $\int_{d \in \mathcal{D}} G(X)(d,d)$ is a wedge of the following diagram.

$$G(Y)(x,x) \xrightarrow{G(Y)(id_x, f)} G(Y)(x,y) \xleftarrow{G(Y)(f, id_y)} G(Y)(y,y)$$

Where the vertices of the cone are constructed by composing the projection map with the action of G on f, i.e., $G(f)(x,x) \circ \pi_x$. By universality, this wedge uniquely factors through the end $\int_{d \in \mathcal{D}} G(Y)(d,d)$. This factorization defines the morphism action $\mathbf{end}(f)$.

$$\begin{array}{lcl} \mathbf{end}(G)(-) & : & \mathcal{C} \to \mathcal{E} \\ \mathbf{end}(G)(x) & \mapsto & \int_{d \in \mathcal{D}} G(x)(d,d) \\ \mathbf{end}(G)(f) & \mapsto & \mathsf{factor}(\int_{d \in \mathcal{D}} G(x)(d,d)) \end{array}$$

An important subtlety here is that $F(X)$ should have an end in $\mathcal{E}$ for every X. In our case, this is a consequence of completeness of $\textsc{Set}_1$.[57] To actually use the functor **end** to define the semantics of universal quantifications, we need to precompose the semantics of its body with the **sift** functor to separate the quantified variable from the remainder of the context.

[57] *See Mac Lane [MacLane, 1971] chapter 9.5 corollary 2.*

$$\begin{array}{lll} \mathsf{sift} & : & (\llbracket \Delta \rrbracket \times \llbracket k \rrbracket)^{op} \times (\llbracket \Delta \rrbracket \times \llbracket k \rrbracket)) \times \llbracket \Phi \rrbracket \\ & & \to ((\llbracket \Delta \rrbracket^{op} \times \llbracket \Delta \rrbracket) \times \llbracket \Phi \rrbracket) \times (\llbracket k \rrbracket^{op} \times \llbracket k \rrbracket) \end{array}$$

We note that **sift** defines an isomorphism in **CAT**.

5.4.3 *On the Existence of Initial Algebras*

In general, it is not the case that any endofunctor has an initial algebra. For certain classes of endofunctors, it can be shown that an initial algebra exists by means of Adámek's theorem [Adámek, 1974]. Here, we present a condensed argument for why we expect that functors interpreting well-formed types of kind $k \rightsquigarrow k$ (for any k) have initial algebras; a more thorough formal treatment of the construction of initial algebras is a subject of further study.

The intuition behind Adámek's construction is that repeated applications of an endofunctor $F : \mathcal{C} \to \mathcal{C}$ converge after infinite iterations, reaching a fixpoint. If $\mathcal{C}$ has an initial object and *ω-colimits*,[58] we can define the initial algebra of F as the ω-colimit of the following chain:

[58] *That is, colimits over diagrams defined as a functor on the thin category generated from the poset of natural numbers.*

$$\bot \xrightarrow{!} F\bot \xrightarrow{F!} FF\bot \xrightarrow{FF!} FFF\bot \xrightarrow{FFF!} \ldots$$

Where $\bot$ is the initial object in $\mathcal{C}$ and $!_X : \bot \to X$ the unique map from $\bot$ to X. A crucial stipulation is that F should be *ω-cocontinuous*, meaning that it preserves ω-colimits.

Thus, for the functors interpreting higher-order types to have an initial algebra, we must argue that all higher-order types are interpreted to a ω-cocontinuous functor. This prompts a refinement of the semantics for kinds discussed in Section 5.4.1, where we impose the additional restriction that the interpretation of a kind of the form $k_1 \rightsquigarrow k_2$ is a ω-cocontinuous functor from $[\![k_1]\!]$ to $[\![k_2]\!]$. Subsequently, we must show that Figure 21 actually inhabits this refined semantics.

Johann and Polonsky [2019] present an inductive argument showing the existence of initial algebras for a universe of higher-kinded data types is similar to our definition of well-formed terms in Figure 17. While their proof establishes the more general property of λ-cocontinuity (for an arbitrary limit ordinal λ) for the functors interpreting higher-kinded types, we expect that the relevant cases of their inductive proof—specifically the cases for products, coproducts, type application, and the μ functor—can be adapted to our setting. What remains is to show that the semantics of type level λ-abstraction and function types is a ω-cocontinuous functor. For λ-abstraction, we transport along the currying isomorphism, which should preserve ω-cocontinuity. For function types, we require that the functor $(-)^X : \textsc{Set} \to \textsc{Set}$ is ω-cocontinuous for all X, which, as Johann and Polonsky [2019] point out, is indeed the case. Expanding this proof sketch into a full proof of the existence of initial algebras is future work.

5.4.4 *Arrow Types Correspond to Morphisms*

To define the semantics of well-typed terms, it is crucial that we can relate arrow types—i.e., of the form $\tau_1 \xrightarrow{k} \tau_2$—to morphisms in the category $[\![k]\!]$. To make this more precise, consider the typing rule for left projections. To define its semantics, we would like to use the cartesian structure of the category $[\![k]\!]$, which implies the existence of a *morphism* $\pi_1 : [\![k]\!](x \times y, x)$ for $x, y \in [\![k]\!]$. However, the rule T-FST implies that π_1 should be related to an *object* in SET_1, i.e., $[\![\tau_1 \times \tau_2 \xrightarrow{k} \tau_1]\!]$. To mediate between morphisms in $[\![k]\!]$ and objects in SET_1 calls for a suitable currying/uncurrying isomorphism for arrow types, though we highlight that the required isomorphism is different from the usual currying isomorphism arising from the existence of right adjoints for the tensor product in closed monoidal catetegories, in the sense that $[\![\tau_1 \xrightarrow{k} \tau_2]\!]$ does not define an internal hom for the objects $[\![\tau_1]\!], [\![\tau_2]\!]$ but rather inernalizes the morphisms between these objects in a *different* category.

Theorem 1. *Given a kind* k*, morphisms of the category* $[\![k]\!]$ *are internalized as objects in* SET_1 *through the following bijection between hom-sets:*

$$[\![k]\!](F(\delta) \times [\![\tau_1]\!](\delta^\circ), [\![\tau_2]\!](\delta)) \simeq \text{SET}_1(F(\delta), [\![\tau_1 \xrightarrow{k} \tau_2]\!](\delta)) \quad (2)$$

Where $\delta \in [\![\Delta]\!]^{op} \times [\![\Delta]\!]$ *and* $\delta^\circ \in ([\![\Delta]\!]^{op} \times [\![\Delta]\!])^{op}$ *its complement, which is defined by swapping the objects representing contravariant respectively covariant occurrences of the variables in* Δ*. Let* $F : [\![\Delta]\!]^{op} \times [\![\Delta]\!] \to \text{SET}_1$ *be a functor. In a slight abuse of notation, we also write* $F(\delta)$ *for the "lifting" of* F *to an object in the (functor) category* $[\![k]\!]$ *that ignores all the additional variables on which* $[\![\tau_1]\!]$ *and* $[\![\tau_2]\!]$ *depend.*

Proof. Figure 22 shows how the isomorphism in Equation (2) is computed. The first step of the derivation rewrites the left-hand side of the isomorphism to a sequence of zero or more ends in the category of very large sets, allowing us to apply currying for exponentials in SET_1 in the subsequent step. This is justified by cartesian closedness of SET, because the objects $[\![\tau_1]\!](\delta^\circ)(x_1)\cdots(x_n)$ and $[\![\tau_2]\!](\delta)(x_1)\cdots(x_n)$ are included in the

$$
\begin{aligned}
& [\![k]\!](F(\delta) \times [\![\tau_1]\!](\delta^\circ), [\![\tau_2]\!](\delta)) \\
= & \int_{x_1 \in [\![k_1]\!]} \cdots \int_{x_n \in [\![k_n]\!]} \textsc{Set}_1(F(\delta) \times [\![\tau_1]\!](\delta^\circ)(x_1)\cdots(x_n), [\![\tau_2]\!](\delta)(x_1)\cdots(x_n)) \\
\simeq & \int_{x_1 \in [\![k_1]\!]} \cdots \int_{x_n \in [\![k_n]\!]} \textsc{Set}_1(F(\delta), [\![\tau_2]\!](\delta)(x_1)\cdots(x_n)^{[\![\tau_1]\!](\delta^\circ)(x_1)\cdots(x_n)}) \\
\simeq & \textsc{Set}_1(F(\delta), \int_{x_1} \cdots \int_{x_n} [\![\tau_2]\!](\delta)(x_1)\cdots(x_n \in [\![k_n]\!])^{[\![\tau_1]\!](\delta^\circ)(x_1)\cdots(x_n)}) \\
\simeq & \textsc{Set}_1(F(\delta), [\![\tau_1 \xrightarrow{k} \tau_2]\!](\delta))
\end{aligned}
$$

Figure 22: Derivation of the isomorphism in Equation (2), where $k = k_1 \rightsquigarrow \cdots \rightsquigarrow k_n \rightsquigarrow \star$

image of the fully faithful inclusion functor I. Next, we use the fact that the covariant hom-functor $\textsc{Set}_1(x, -)$ is continuous and thus preserves ends:[59]

[59] *See MacLane [1971], page 225 Equation 4.*

$$
\int_{y \in \mathcal{C}} \textsc{Set}_1(x, G(y,y)) \quad \simeq \quad \textsc{Set}_1(x, \int_{y \in \mathcal{C}} G(y,y)) \qquad (3)
$$

By repeatedly applying the identity above, we can distribute the aforementioned sequence of ends over the functor $\textsc{Set}_1(F(\delta), -)$. Intuitively, this corresponds to distributing universal quantification over logical implication in the scenario that the quantified variable does not occur freely in the antecedent, which is axiomatized in some flavors of first-order logic, though we apply a much more general instance of the same principle here. The final step then follows from the standard definition of η-equivalence implied by cartesian closedness of **CAT**. $\square$

We write $\uparrow(-)/\downarrow(-)$ for the functions that transport along the isomorphism defined in Equation (2).

5.4.5 *Interpreting Terms*

Well-typed terms, of the form $\Gamma \vdash M : \sigma$, are interpreted as natural transformations from the interpretation their context, $[\![\Gamma]\!]$, to the interpretation of their type, $[\![\sigma]\!]$. At component $\delta \in$

$$
\begin{aligned}
[\![\Gamma \vdash x : \sigma]\!]_\delta &= \textbf{lookup}^{\Gamma}_{x} \\
[\![\Gamma \vdash M\ N : \tau_2]\!]_\delta &= \mathsf{eval} \circ \langle [\![\Gamma \vdash M : \tau_1 \Rightarrow \tau_2]\!]_\delta, [\![\Gamma \vdash N : \tau_1]\!]_\delta \rangle \\
[\![\Gamma \vdash \lambda x.M : \tau_1 \Rightarrow \tau_2]\!]_\delta &= \mathsf{curry}([\![\Gamma, x : \tau_1 \vdash M : \tau_2]\!]_\delta) \\
[\![\Gamma \vdash \textbf{let}\ (x : \sigma_1) = M\ \textbf{in}\ N : \sigma_2]\!]_\delta &= \mathsf{eval} \circ \langle \mathsf{curry}([\![\Gamma, x : \sigma_1 \vdash N : \sigma_2]\!]_\delta), [\![\Gamma \vdash N : \sigma_1]\!]_\delta \rangle \\
[\![\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma]\!]_\delta &= [\![\Gamma \vdash M : \sigma]\!]_\delta \quad \text{(isomorphic per Equation (3))} \\
[\![\Gamma \vdash M@\tau : \sigma[\tau/\alpha]]\!]_\delta &= \pi_{[\![\tau]\!]} \circ [\![\Gamma \vdash M : \forall\alpha.\sigma]\!]_\delta \\
[\![\Gamma \vdash \textbf{in} : \tau\ \mu(\tau) \xrightarrow{k} \mu(\tau)]\!]_\delta &= \uparrow(\mathsf{in} \circ \pi_2) \\
[\![\Gamma \vdash \textbf{unin} : \mu(\tau) \xrightarrow{k} \tau\ \mu(\tau)]\!]_\delta &= \uparrow(\mathsf{unin} \circ \pi_2) \\
[\![\Gamma \vdash \textbf{map}\langle M\rangle^{\tau} : \tau\ \tau_1 \xrightarrow{k_2} \tau\ \tau_2]\!]_\delta &= \uparrow(\lambda(\gamma, x).[\![\tau]\!](\delta)(\lambda y.\downarrow([\![\Gamma \vdash M : \tau_1 \xrightarrow{k_1} \tau_2]\!]_\delta)(\gamma, y))) \\
[\![\Gamma \vdash \llparenthesis\ M\ \rrparenthesis^{\tau_1} : \mu(\tau_1) \xrightarrow{k} \tau_2]\!]_\delta &= \\
&\uparrow(\lambda(\gamma, x).\mathsf{cata}(\lambda y.\downarrow([\![\Gamma \vdash M : \tau_1\ \tau_2 \xrightarrow{k} \tau_2]\!]_\delta)(\gamma, y))) \\
[\![\Gamma \vdash \boldsymbol{\pi_1} : \tau_1 \times \tau_2 \xrightarrow{k} \tau_1]\!]_\delta &= \uparrow(\pi_1 \circ \pi_2) \\
[\![\Gamma \vdash \boldsymbol{\pi_2} : \tau_1 \times \tau_2 \xrightarrow{k} \tau_2]\!]_\delta &= \uparrow(\pi_2 \circ \pi_2) \\
[\![\Gamma \vdash M \blacktriangle N : \tau \xrightarrow{k} \tau_1 \times \tau_2]\!]_\delta &= \\
&\uparrow(\ \langle\ \downarrow([\![\Gamma \vdash M : \tau \xrightarrow{k} \tau_1]\!]_\delta)\ ,\ \downarrow([\![\Gamma \vdash N : \tau \xrightarrow{k} \tau_2]\!]_\delta)\ \rangle\ ) \\
[\![\Gamma \vdash \boldsymbol{\iota_1} : \tau_1 \xrightarrow{k} \tau_1 + \tau_2]\!]_\delta &= \uparrow(\iota_1 \circ \pi_2) \\
[\![\Gamma \vdash \boldsymbol{\iota_2} : \tau_2 \xrightarrow{k} \tau_1 + \tau_2]\!]_\delta &= \uparrow(\iota_2 \circ \pi_2) \\
[\![\Gamma \vdash M \blacktriangledown N : \tau_1 + \tau_2 \xrightarrow{k} \tau]\!]_\delta &= \\
&\uparrow(\ [\ \downarrow([\![\Gamma \vdash M : \tau_1 \xrightarrow{k} \tau]\!]_\delta)\ ,\ \downarrow([\![\Gamma \vdash N : \tau_2 \xrightarrow{k} \tau]\!]_\delta)\ ]\ ) \\
[\![\Gamma \vdash \textbf{tt} : \mathbb{1}]\!]_\delta &= !\quad \text{(the unique morphism to the terminal object)} \\
[\![\Gamma \vdash \textbf{absurd} :\ \Rightarrow \tau]\!]_\delta &= \mathsf{curry}(h \circ \pi_2)
\end{aligned}
$$

Figure 23: Semantics of Well-Typed Terms

$[\![\Delta]\!]^{\mathrm{op}} \times [\![\Delta]\!]$ this transformation is given by a function with the following type:

$$[\![\ \Gamma \vdash M : \sigma\ ]\!]_\delta : [\![\ \Gamma\ ]\!](\delta) \to [\![\ \sigma\ ]\!](\delta)$$

Here, $[\![\Gamma]\!]$ is defined componentwise by mapping contexts to a left-associated product of its elements, analogous to how we defined the interpretation of kind contexts in Section 5.4.1. Figure 23 shows the interpretation of well-typed terms in its entirety.

The interpretation of λ-abstraction and application is defined in terms of the cartesian closed structure of SET, which is preserved by its inclusion in SET_1. For a type abstractions of the form $\Lambda\alpha.M$, its semantics follows from the fact that hom-functors preserves ends (see Equation (3)), which implies a bijection between the set of morphisms that interprets the type abstraction and the set of morphisms into which we interpret its body. We remark that this only works because α does not occur free in Γ, meaning that we know that $[\![\Gamma\ ]\!]$ does not depend on α in $[\![\Gamma \vdash M : \sigma]\!]_{\delta,(\alpha,\alpha)} : [\![\Gamma]\!](\delta,(\alpha,\alpha)) \to [\![\sigma]\!](\delta,(\alpha,\alpha))$, and thus we can view $[\![\Gamma]\!]$ as a constant when applying the isomorphism. The semantics of a type application $M\ @\tau$ is then given by the projection map at component $[\![\tau]\!]$ of the end interpreting the type of M. For the introduction and elimination forms of (co)product types, and the unit and empty type, we define the semantics in terms of the corresponding (co)limits in SET_1, applying the currying isomorphism defined in Equation (2) to mediate with arrow types. Similarly, a semantics for the mapping and folding primitives also follows from the currying isomorphism defined in Equation (2).

Both the denotation function $[\![-]\!]$ as well as the function it computes are total. Consequently, a well-typed value can be computed from every well-typed term. In this sense, the categorical model provides us with a sound computational model of the calculus, which we could implement by writing a definitional interpreter [**?**]. In the next section, we will discuss how a more traditional small-step operational semantics can be derived from the same categorical model.

5.5 OPERATIONAL SEMANTICS

The previous section gave an overview of a categorical semantics of our calculus. In this section, we define a small-step operational semantics for our calculus, and discuss how it relates to the categorical model.

$$
\begin{array}{rcll}
v & := & \lambda x.M \mid \Lambda\alpha.M \mid \mathbf{in}\ \colorbox{lightgray}{$\overline{\tau}\,v$} \mid \mathbf{unin}\ \colorbox{lightgray}{$\overline{\tau}\,v$} \mid (v_1 \blacktriangle v_2)\ \colorbox{lightgray}{$\overline{\tau}\,v$} & \text{(Values)} \\
 & \mid & \iota_1\ \colorbox{lightgray}{$\overline{\tau}\,v$} \mid \iota_2\ \colorbox{lightgray}{$\overline{\tau}\,v$} \mid \mathbf{map}\langle v\rangle^{\tau'}\ \colorbox{lightgray}{$\overline{\tau}$} \mid (\!|\, v \,|\!)^{\tau'}\ \colorbox{lightgray}{$\overline{\tau}$} \mid \pi_1\ \colorbox{lightgray}{$\overline{\tau}$} \mid \pi_2\ \colorbox{lightgray}{$\overline{\tau}$} & \\
 & \mid & (v_1 \blacktriangledown v_2)\ \colorbox{lightgray}{$\overline{\tau}$} \mid \mathbf{tt}\ \colorbox{lightgray}{$\overline{\tau}$} \mid \mathbf{absurd}\ \colorbox{lightgray}{$\overline{\tau}\,v$} & \\
 & & & \\
E & := & [] \mid E\ M \mid v\ E \mid E\ \tau \mid \mathbf{let}\ (x : \sigma) = E\ \mathbf{in}\ M \mid \mathbf{let}\ (x : \sigma) = v\ \mathbf{in}\ E & \text{(Contexts)} \\
 & \mid & \mathbf{map}\langle E\rangle^{\tau} \mid (\!|\, E \,|\!)^{\tau} \mid E \blacktriangle M \mid v \blacktriangle E \mid E \blacktriangledown M \mid v \blacktriangledown E &
\end{array}
$$

Figure 24: Values and Evaluation Contexts. Highlights indicate optional occurrences of (type) arguments.

5.5.1 *Reduction Rules*

We define our operational semantics as a reduction semantics in the style of Felleisen and Hieb [1992]. Figure 24 shows the definition of values and evaluation contexts. In our definition of values, we must account for the fact that language primitives can exist at any kind. For example, the primitive ι_1 by itself is a value of type $\tau_1 \xrightarrow{k} \tau_1 + \tau_2$. Simultaneously, applying ι_1 with a value and/or a sequence of type arguments (the number of which depends on the kind of its arrow type), also yields a value. In fact, all the *partial applications* of ι_1 with only some of its type arguments, or all type arguments but no value argument, are also values. We use gray highlights to indicate such an optional application with type and/or value arguments in the definition of values.

Figure 25 defines the reduction rules. We split the rules in two categories: the first set describes β-reduction[60] for the various type formers, while the second set determines how the $\mathbf{map}\langle - \rangle^{-}$ primitive computes. Similar to the definition of values and contexts in Figure 24, we use the notation $\overline{\tau}$ to depict a sequence of zero or more type applications. Unlike for values, these type arguments are not optional; terms typed by an arrow types must be fully applied with all their type arguments before they reduce. The notation $N \bullet M$ is used as a syntactic shorthand for the composition of two arrow types, which is defined through η-expansion of all its type arguments and the term argument. The reduction rules for the $\mathbf{map}\langle\tau\rangle^{M}$ primitive are type directed, in the sense that the selected reduction depends on τ. This is

[60] *Here, we mean "β-reduction" in the more general sense of simplifying an application of an elimination form to an introduction form.*

$$
\begin{array}{rcll}
((\lambda x.M)\ v & \longrightarrow & M[v/x] & (1)\\
\mathbf{let}\ (x:\sigma) = v\ \mathbf{in}\ M & \longrightarrow & M[v/x] & (2)\\
(\Lambda\alpha.M)\ \tau & \longrightarrow & M[\tau/\alpha] & (3)\\
\mathbf{unin}\ \overline{\tau}\ (\mathbf{in}\ \overline{\tau}\ v) & \longrightarrow & v & (4)\\
(\!|\ v_1\ |\!)^{\tau'}\ \overline{\tau}\ (\mathbf{in}\ \overline{\tau}\ v_2) & \longrightarrow & v_1\ \overline{\tau}\ (\mathbf{map}\langle (\!|\ v_1\ |\!)^{\tau'} \rangle^{\tau'}\ \overline{\tau}\ v_2) & (5)\\
\pi_1\ \overline{\tau}\ ((v_1\ \blacktriangle\ v_2)\ \overline{\tau}\ v) & \longrightarrow & v_1\ \overline{\tau}\ v & (6)\\
\pi_2\ \overline{\tau}\ ((v_1\ \blacktriangle\ v_2)\ \overline{\tau}\ v) & \longrightarrow & v_2\ \overline{\tau}\ v & (7)\\
(v_1\ \blacktriangledown\ v_2)\ \overline{\tau}\ (\iota_1\ \overline{\tau}\ v)) & \longrightarrow & v_1\ \overline{\tau}\ v & (8)\\
(v_1\ \blacktriangledown\ v_2)\ \overline{\tau}\ (\iota_2\ \overline{\tau}\ v)) & \longrightarrow & v_2\ \overline{\tau}\ v & (9)\\
 & & & \\
\mathbf{map}\langle v_1 \rangle^{(\lambda X.X)}\ \overline{\tau}\ v_2 & \longrightarrow & v_1\ \overline{\tau}\ v_2 & (10)\\
\mathbf{map}\langle v_1 \rangle^{\mu(\tau')}\ \overline{\tau}\ (\mathbf{in}\ \overline{\tau}\ v_2) & \longrightarrow & \mathbf{in}\ \overline{\tau}\ (\mathbf{map}\langle v_1 \rangle^{(\tau'\ \mu(\tau'))}\ \overline{\tau}\ v_2) & (11)\\
\mathbf{map}\langle v \rangle^{\tau_1 \times \tau_2}\ \overline{\tau}\ ((v_1\ \blacktriangle\ v_2)\ \overline{\tau}\ v_3) & & & \\
\multicolumn{3}{r}{\longrightarrow ((\mathbf{map}\langle v \rangle^{\tau_1} \bullet v_1)\ \blacktriangle\ (\mathbf{map}\langle v \rangle^{\tau_2} \bullet v_2))\ \overline{\tau}\ v_3} & (12)\\
\mathbf{map}\langle v_1 \rangle^{\tau_1 + \tau_2}\ \overline{\tau}\ (\iota_1\ \overline{\tau}\ v_2) & \longrightarrow & \iota_1\ \overline{\tau}\ (\mathbf{map}\langle v_1 \rangle^{\tau_1}\ \overline{\tau}\ v_2) & (13)\\
\mathbf{map}\langle v_1 \rangle^{\tau_1 + \tau_2}\ \overline{\tau}\ (\iota_2\ \overline{\tau}\ v_2) & \longrightarrow & \iota_2\ \overline{\tau}\ (\mathbf{map}\langle v_1 \rangle^{\tau_2}\ \overline{\tau}\ v_2) & (14)\\
\mathbf{map}\langle v \rangle^{\mathbf{1}}\ \overline{\tau}\ (\mathbf{tt}\ \overline{\tau}) & \longrightarrow & \mathbf{tt}\ \overline{\tau} & (15)
\end{array}
$$

$$N \bullet M \quad \triangleq \quad \overline{\Lambda\alpha}.\lambda x.N\ \overline{\alpha}\ (M\ \overline{\alpha}\ x)$$

Figure 25: Reduction rules

necessary, because in an application of $\mathbf{map}\langle - \rangle^{-}$ to a value, there is no way to decide whether to apply the function or to push the $\mathbf{map}\langle - \rangle^{-}$ further inwards by only looking at the value.

5.5.2 *Relation to the Denotational Model*

The reduction rules shown in Figure 25 define a computational model for our calculus. We now discuss how this model arises from the denotational model discussed in Section 5.4. Informally speaking, reducing a term should not change its meaning. This intuition is reflected by the following implication, which states if M reduces N, their semantics should be equal.[61]

[61] *This property implies what Devesas Campos and Levy [2018] call* soundness *of the denotational model with respect to the operational model. Their soundness property is about a big-step relation; ours is small-step.*

$$M \longrightarrow N \implies [\![M]\!] = [\![N]\!] \tag{4}$$

While we do not give a formal proof of the implication above, by relying on the categorical model to inform how terms compute we can be reasonably confident that our semantics does not contain any reductions that violate this property. That is, all the reductions shown in Figure 25 are supported by an equality of morphisms in the categorical model.

What does this mean, specifically? The semantics of well-typed terms is given by a natural transformation, so if $M \longrightarrow N$, M and N should be interpreted as the same natural transformation. Equivalence of natural transformations is defined pointwise in terms of the equality relation for morphisms in the underlying category. In our case, this is the category Set, as terms are interpreted as natural transformations between functors into Set. By studying the properties—expressed as equalities between morphisms—of the constructions that give a semantics to the different type formers, and reifying these equalities as syntactic reduction rules, we obtain an operational model that we conjecture respects the denotational model by construction.

Let us illustrate this principle with a concrete example. The semantics of a sum type $\tau_1 + \tau_2 : k$ is given by a coproduct in the category $[\![k]\!]$. The universal property of coproducts tells us that $[f, g] \circ \iota_1 = f$ and $[f, g] \circ \iota_2 = g$, or in other words, constructing and then immediately deconstructing a coproduct is the same as doing nothing. Rules (8) and (9) in Figure 25 reflect these equations. That is, since the ι_1, ι_2, and $- \blacktriangledown -$ primitives are interpreted as the injections ι_1, ι_2, and unique morphism $[-,-]$ respectively, the universal property of coproducts tells us that the left-hand side and right-hand side of rule (8) and (9) in Figure 25 are interpreted to equal morphism in the categorical domain.

The remaining reduction rules are justified by the categorical model in a similar fashion. More specifically:

- Rules (1,2) follow from the β-law for exponential objects, which states that $\mathsf{eval} \circ \langle \mathsf{curry}(f), id \rangle = f$.

- Rule (3) holds definitionally, assuming type substitution is appropriately defined such that it corresponds to functor application.
- Rule (4) follows from Lambek's lemma, which states that the component of an initial algebra is always an isomorphism. That is, there exists a morphism unin such that $\mathsf{unin} \circ \mathsf{in} = id$.
- Rule (5) reflects the universal property of folds, i.e., $\mathsf{cata}(f) \circ \mathsf{in} = f \circ F(\mathsf{cata}(f))$.
- Rules (6,7) follow from the universal property of products, which states that $\pi_1 \circ \langle f, g \rangle = f$ and $\pi_2 \circ \langle f, g \rangle = g$.
- Rule (10) mirrors the identity law for functors, i.e. $F(id) = id$.
- Rule (11) is derived from naturality of the component of the initial algebra of higher-order functors, which states that $\mu(F)(f) \circ \mathsf{in} = \mathsf{in} \circ F(\mu(F))(f)$.
- Rule (12,13,14,15) are derived from the way (co)-limits are computed pointwise in functor categories. For example, the morphism action of the product of two functors F and G is defined as $(F \times G)(f) = \langle F(f) \circ \pi_1, G(f) \circ \pi_2 \rangle$, which gives rise to rule (12).

5.6 RELATED WORK

The problem of equipping functional languages with better support for modularity as been studied extensively in the literature. One of the earlier instances is the *Algebraic Design Language* (ADL) by Kieburtz and Lewis [1995], which features language primitives for specifying computable functions in terms of algebras. ADL overlaps to a large extent with the first-order fragment of our calculus, but lacks support for defining nested data types. Zhang et al. [2021] recently proposed a calculus and language for *compositional programming*, called CP. Their language design is inspired by *object algebras*, which in turn is based on the *tagless* *final* approach [Carette et al., 2009a, Kiselyov, 2010] and *final algebra semantics* [Wand, 1979], which, according to [Wand, 1979, §7], is an extension of *initial algebra semantics*. These lines of work thus provide similar modularity as initial algebra semantics, but in a way that does not require *tagged values*. While the categorical foundations of Zhang et al.'s CP language seems to be an open question, the language provides flexible support for modular programming, in part due to its powerful notion of subtyping. We are not aware of attempts to model (higher-order) effects and handlers using CP. In contrast, our calculus is designed to have a clear categorical semantics. This semantics makes it straightforward to define state of the art type safe modular (higher-order) effects and handlers. Morris and McKinna [2019a] define a language that has built-in support for *row types*, which supports both extensible records and variants. While their language captures many known flavors of extensibility, due to parameterizing the type system over a so-called *row theory* describing how row types behave under composition, rows are restricted to first order types. Consequently, they cannot describe any modularity that hinges on the composition of (higher-order) signature functors.

The question of including nested data types in a language's support for modularity has received some attention as well. For example, Cai et al. [2016] develop an extension of F_ω with equirecursive types tailored to describe patterns from datatype generic programming. Their calculus is expressive enough to capture the modularity abstractions discussed in this paper, including those requiring nested data types, but lacks a denotational model; a correspondence between a subset of types in their calculus and (traversable) functors is discussed informally. Similarly, Abel et al. [2005] consider an operational perspective of traversals over nested datatypes by studying several extensions of F_ω with primitives for *(generalized) Mendler iteration and coiteration*. Although these are expressive enough to describe modular higher-order effects and handlers, their semantic foundation is very different from the semantics of the primitive fold operation in our calculus. It is future work to investigate how our calculus can be extended with support for codata.

A major source of inspiration for the work in this paper are recent works by Johann and Polonsky [2019], Johann et al. [2021], and Johann and Ghiorzi [2021], which respectively study the semantics and parametricity of nested data types and GADTs. For the latter, the authors develop a dedicated calculus with a design and semantics that is very similar to ours. Still, there are some subtle but key differences between the designs; for example, their calculus does not include general notions of ∀-types and function types, but rather integrates these into a single type representing natural transformations between type constructors. While their setup does not require the same stratification of the type syntax we adopt here, it is also slightly less expressive, as the built-in type of transformations is restricted to closing over 0-arity arguments.

Data type generic programming commonly uses a *universe of descriptions* [Benke et al., 2003], which is a data type whose inhabitants correspond to a signature functor. Generic functions are commonly defined by induction over these descriptions, ranging over a semantic reflection of the input description in the type system of a dependently-typed host language [Dagand, 2013a]. In fact, Chapman et al. [2010a] considered the integration of descriptions in a language's design by developing a type theory with native support for generic programming. We are, however, not aware of any notion of descriptions that corresponds to our syntax of well-formed types.

5.7 CONCLUSION AND FUTURE WORK

In this paper, we presented the design and semantics of a calculus with support for modularity. We demonstrated it can serve as a basis for capturing several well-known programming patterns for retrofitting type-safe modularity to functional languages, such as modular interpreters in the style of Data Types à la Carte, and modular (higher-order) algebraic effects. The formal semantics associates these patterns with their motivating concepts, creating the possibility for a compiler to benefit from their properties such as by performing fusion-based optimizations.

POSTSCRIPT

This chapter contributed a core calculus that nicely captures the essence of how extensible data types can be incorporated in the design of a functional meta language in a principled way. How do we connect this result to the overarching goal of enabling language designers to define reusable programming language components as intrinsically-typed definitional interpreters?

There are several steps to take. First and foremost, the results should be extended to be able to describe modularity of *indexed data types*, as discussed in Section 2.2, to allow us to encode intrinsically-typed interpreters in the language. This introduces additional challenges, as functions that eliminate indexed data types locally rely on additional equalities to rule out redundant cases for ill-typed inputs. Beyond that, a key selling point of intrinsically-typed definitional interpreters is their readability, which is mostly due to *dependent pattern matching* [Cockx, 2017]. A meta language design should include this, and other syntactical conveniences, to be a feasible option for defining reusable programming language components.

CONCLUSIONS

6 CONCLUSIONS

The core problem we addressed in this thesis is the high development cost associated with the development of programming languages with formally specified type systems and verified type soundness. To reduce this cost, we pinpointed two areas in which it is necessary to make progress:

1. the development of semantic techniques that allow us to specify the semantics of language constructs in a way that is independent of the context (i.e., language) in which they are used, and

2. the design of (functional) meta-languages that support modularity out-of-the-box, that can be used to define modular and reusable language components without having to deal with the additional overhead that is usually incurred by modularity.

In the remainder of this chapter, we will revisit the contributions of this thesis, and reflect on these contributions in light of the hypotheses from Section 1.2.

6.1 SUMMARY OF THE CONTRIBUTIONS

Each chapter in this thesis contains its own contributions. Here, we summarize the core contributions of this thesis.

In Chapter 2, we showed how to embed modular intrinsically-typed definitional interpreters in Agda, and developed a notion of *intrinsically-typed language fragments*, that form a self-contained, type safe, and modular specification of a language construct. Language fragments can be freely composed and reused to build bigger programming languages in a way that all these properties are preserved.

In Chapter 3, we defined abstractions for defining *modular semantics for higher-order effects*, and developed reasoning infrastructure to prove these semantics correct with respect to equational theories that give an axiomatic specification of an effect's intended behavior. This enables us to give a modular semantics to language constructs such as λ-abstraction or exception catching, and reason about the correctness of these semantics.

In Chapter 4, we developed the design of a domain-specific meta language for defining reusable programming language components. Key features of the design are an *effect system that supports higher-order effects*, and *native support for modular data types*. As a result, the language allows us to write modular interpreters (effectively solving the *expression problem* [Wadler, 1998]) without having to resort to shallow embeddings of initial algebra semantics.

Finally, in Chapter 5, we presented a core calculus with *built-in primitives for modularity*, together with a type system and semantics. The calculus is expressive enough to capture many familiar programming abstractions for modularity, such as modular data types in the style of Data Types á la Carte [Swierstra, 2008], or inductively-defined free monads. The calculus provides a formal basis for understanding languages with built-in support for modular data types, such as the language design from Chapter 4.

6.2 HYPOTHESIS 1: INTRINSICALLY-TYPED INTERPRETERS

In part I of this thesis (Chapters 2 and 3), we explored the following hypothesis:

> reusable programming language components should be concise, readable, and safe-by-construction. Intrinsically-typed definitional interpreters are an excellent match for these requirements.

In Chapter 2, we developed techniques for writing modular intrinsically-typed definitional interpreters. Crucially, we pinpointed *canonicity-preserving* separation and inclusion predicates (Section 2.3) that witness respectively type-safe union and extension of value domains. These turned out to be the key ingredients for defining intrinsically-typed interpreters in a way that they can be composed in a safe-by-construction manner.

This is, however, a mere first step towards leveraging intrinsically-typed definitional interpreters for the specification of reusable programming language components. As we pointed out in Section 2.5, the abstractions developed in Chapter 2 necessarily presuppose both the notion of typing and semantics used to define a language fragment. Although the techniques we developed for modularizing intrinsically-typed definitional interpreters can be generalized to languages with lexical binding that denote into a more expressive semantic domain (Section 2.5), this leaves the question of whether the techniques apply to language components that do not fit this abstraction. Beyond that it is also still an open question whether language fragments could be generalized even further to allow the language designer to choose a flavor of typing and semantics that fits their language project.

ANOTHER QUESTION LEFT open at the end of Chapter 2 is how to adequately deal with side effects of the interpreter. In the setup used in Section 2.5, we make the assumption that a suitable monad exists that encapsulates all the required side effects for denoting a particular combination of language fragment. Depending on the combination of effects, constructing this monad may actually be tricky. Specifically, if we require higher-order effects and/or *intrinsically-typed effects* [62] to denote a language, existing treatments of modular effects are insufficient.

[62] *An example of a monad modelling intrinsically-typed effects is the monad used by Bach Poulsen et al. [2018] to denote a language with polymorphic references.*

In Chapter 3, we give a partial answer to this question, by developing a unified approach for defining modular semantics for higher-order effects by elaborating them to algebraic effects. While the approach solves the modularity problem for higher-order effects, it is not equipped to deal with intrinsically-typed effects yet. As a result, we cannot yet use it to define the monad(s) assumed in Section 2.5 in a modular fashion. Most likely, this would require us to generalize the current approach, where we construct monad morphisms between different free monads in the category of Agda sets, to categories of monotone predicates to add the necessary *world-indexing* for modelling intrinsically-typed effects such as well-typedness of polymorphic references.

All in all, we conclude that the contributions presented in Chapters 2 and 3 provide us with a solid basis for understanding modularity of intrinsically-typed interpreters and higher-order effects. Within the current state of affairs we can already construct reusable language components for many different language constructs by writing intrinsically-typed definitional interpreters. However, before we can scale the approach to more advanced type systems, such as the interpreters for imperative languages by Bach Poulsen et al. [2018] or the interpreter for linear session-typed languages by Rouvoet et al. [2020], there are some open research questions to be addressed first. Even more expressive systems, such as System F and dependent type theory, remain out of reach until the intrinsically-typed approach has been extended to these languages.

6.3 HYPOTHESIS 2: META LANGUAGE DESIGN

In part II (Chapters 4 and 5) of this thesis, we explored the following hypothesis:

> modularity adds significant syntactic and interpretative overhead when using state-of-the art (dependently-typed) programming languages to define reusable programming language components. Incorporating modular inductive data types in the design of functional languages is therefore a essential first step in the development of meta languages for the purpose of defining reusable programming language components.

In Chapter 4, we presented a meta language design for defining reusable programming language components, that features built-in support for modular data types. The language supports the definition of modular interpreters, solving the expression problem [Wadler, 1998], without incurring the usual syntactic overhead associated with the encodings of initial algebra semantics that are typically used to retrofit this kind of type-safe modularity to functional languages. However, the language lacks a formal specification of its type system and semantics.

In Chapter 5, we worked towards addressing this shortcoming by developing a calculus with built in support for modularity, together with a type system and semantics. While this does allow us to encode many common programming abstractions for modularity in a type-safe manner (including the kind of language components we defined in Chapter 4), there is still a gap between the languages. To close this gap we would need to, for example, define a desugaring from the meta language defined in Chapter 4 to the calculus defined in Chapter 5. Furthermore, the modular data types in these languages are not expressive enough to capture the modular interpreters discussed in Chapter 2. This would require the design of a meta language with built-in support for modular *indexed data types*.

6.4 FUTURE WORK

This thesis presented contributions towards making formally verified language components more accessible, with the goal of making formally specified and verified programming languages attainable for a broader audience of language engineers. To get there, however, there are still some challenges that need to addressed.

6.4.1 *Modular semantics*

When revisiting the first hypothesis in Section 6.2, we already hinted at the open challenges when it comes to using intrinsically-typed definitional interpreters to specify verified and reusable programming language components. In particular, we would need to improve the expressivity of the approach by further generalizing both the notion of typing as well as the semantic domain of language fragments.

To generalize the notion of typing permitted, generalizations of effect systems such as the one by Mycroft et al. [2016] seem like a good starting point. This would, however, require us to further the state-of-the-art in intrinsically-typed definitional interpreters to this class of typings, before we can think about extending the definition of language fragments accordingly. Furthermore, this induces the question of how to compose fragments that use different typing. While it is conceivable that we could leverage similar separation and inclusion predicates as we used to capture assumptions about canonicity to phrase a fragment's typing in terms of assumptions (rather than fixing it upfront), we have yet to explore this direction.

Wile the above concerns *static effects*, similar modularity concerns arise when considering *dynamic effects*. To give a semantics to languages with polymorphic references, for example, we denote into a category of monotone predicates over the memory state. Other language fragments, however, may maintain different invariants, and it is still an open question how we would mediate the different requirements on the semantic domain imposed by such fragments.

The setup in Section 2.5 already hints at a possible solution direction for this problem by factoring all effects into a monadic interface. Behind the scenes, for the languages that have polymorphic references, this monad is implemented by constructing an inductively-defined free monad in a category of monotone predicates. Similarly, the intrinsically-typed interpreter for session-typed λ-calculus by Rouvoet et al. [2020] relies on a inductively defined free monad in the category of predicates over (runtime) session types. Finding a general and modular type of command tree [Hancock and Setzer, 2000] that subsumes these and other intrinsically-typed dynamic effects would be a crucial step in achieving a modular treatment of effects.

6.4.2 *Meta Language Design*

When it comes to meta language design, the languages presented in Chapter 4 and Chapter 5 occupy two very different points in the design space. Where the former works backwards from the programmer's perspective and focusses more on usability, the second puts more emphasis on mathematical rigor and a formal specification. This leaves use with a clear goal of combining these two different perspectives in a single language design. We could get there by adding a formal specification of a type system to the language presented in Chapter 4, or alternatively define a desugaring into the calculus presented in Chapter 5.

Beyond that, when it comes to usability there are more language features we could consider adding which do not currently appear in either language. For example, *functor subtyping* or *row typing* are obvious candidate features that would improve usability.

6.4.3 *Connecting the Dots*

Aside from the work that is still to be done on the individual parts, we may look ahead and think about how to connect the contributions of the two parts in this thesis. After all, Chapters 2 and 3 are concerned with developing the semantic tools necessary to describe reusable language components as intrinsically-typed definitional interpreters, whereas Chapters 4 and 5 investigate the design of meta languages that support the modular definition of untyped definitional interpreters.

The vision that connects these lines of work is to develop a dedicated meta language for developing reusable programming language components by writing intrinsically-typed definitional interpreters. Such a language would allow us write intrinsically-typed definitional interpreters much like we write them in Agda today, but with the key difference that they are modular and composable. Before we are ready to embark on designing such a language, however, we must have a solid understanding of the semantics of composing intrinsically-typed definitional interpreters, as well as how to incorporate modularity for data types and their operations in a language's design and type system. In this thesis, we have laid the necessary foundation, meaning that the time is ripe to start working towards making this vision a reality.

APPENDIX

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. URL https://doi.org/10.1007/3-540-36576-1_2.

Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005a. URL https://doi.org/10.1016/j.tcs.2005.06.002.

Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005b. URL https://doi.org/10.1016/j.tcs.2005.06.002.

Andreas Abel and Ralph Matthes. (co-)iteration for higher-order nested datatypes. 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 1–20. Springer, 2002. URL https://doi.org/10.1007/3-540-39185-1_1.

Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Generalized iteration and coiteration for higher-order nested datatypes. 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 54–69. Springer, 2003. URL https://doi.org/10.1007/3-540-36576-1_4.

Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.*, 333(1-2):3–66, 2005. URL https://doi.org/10.1016/j.tcs.2004.10.017.

Jiří Adámek. Free algebras and automata realizations in the language of categories. *Commentationes Mathematicae Universitatis Carolinae*, 15(4):589–602, 1974.

Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references A stratified semantics of general references. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, page 75. IEEE Computer Society, 2002. URL https://doi.org/10.1109/LICS.2002.1029818.

Amal Jamil Ahmed. *Semantics of Types for Mutable State*. PhD thesis, USA, 2004. AAI3136691.

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proc. ACM Program. Lang.*, 2(ICFP):90:1–90:30, 2018. URL https://doi.org/10.1145/3236785.

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *J. Funct. Program.*, 31:e22, 2021. URL https://doi.org/10.1017/S0956796820000076.

Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016. doi: 10.1145/2837614.2837638. URL https://doi.org/10.1145/2837614.2837638.

Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris. Indexed containers. *J. Funct. Program.*, 25, 2015. URL `https://doi.org/10.1017/S095679681500009X`.

Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. 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 666–679. ACM, 2017. URL `https://doi.org/10.1145/3009837.3009866`.

Michael A. Arbib and Ernest G. Manes. *Arrows, Structures, and Functors: The Categorical Imperative*. Academic Press, 1975.

Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. URL `https://doi.org/10.1145/3209108.3209189`.

Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*, 1999.

Steve Awodey. *Category Theory*. Oxford Logic Guides. Oxford University Press, Oxford, New York, second edition, second edition edition, June 2010. ISBN 978-0-19-923718-0.

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 Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005. URL `https://doi.org/10.1007/11541868_4`.

Casper Bach Poulsen and Cas van der Rest. Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proc.*

ACM Program. Lang., 7(POPL):1801–1831, 2023. URL https://doi.org/10.1145/3571255.

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2(POPL):16:1–16:34, 2018. URL https://doi.org/10.1145/3158104.

Casper Bach Poulsen, Cas van der Rest, and Tom Schrijvers. Staged effects and handlers for modular languages with abstraction. In *Workshop on Partial Evaluation and Program Manipulation (PEPM). http://casperbp. net/store/staged-effects-and-handlers.pdf*, 2021.

Patrick Bahr. Composing and decomposing data types: a closed type families implementation of data types à la carte. In José Pedro Magalhães and Tiark Rompf, editors, *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, pages 71–82. ACM, 2014. URL https://doi.org/10.1145/2633628.2633635.

Patrick Bahr and Tom Hvitved. Compositional data types. In Jaakko Järvi and Shin-Cheng Mu, editors, *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 83–94. ACM, 2011. URL https://doi.org/10.1145/2036918.2036930.

Patrick Bahr and Tom Hvitved. Parametric compositional data types. 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 3–24, 2012a. URL https://doi.org/10.4204/EPTCS.76.3.

Patrick Bahr and Tom Hvitved. Parametric compositional data types. 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 3–24, 2012b. URL https://doi.org/10.4204/EPTCS.76.3.

Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Log. Methods Comput. Sci.*, 10(4), 2014. URL https://doi.org/10.2168/LMCS-10(4:9)2014.

Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, 84(1): 108–123, 2015. URL https://doi.org/10.1016/j.jlamp.2014.02.001.

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

Birthe van den Berg and Tom Schrijvers. A framework for higher-order effects & handlers. *CoRR*, abs/2302.01415, 2023. URL https://doi.org/10.48550/arXiv.2302.01415.

Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. Latent effects for reusable language components. In Hakjoo Oh, editor, *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings*, volume 13008 of *Lecture Notes in Computer Science*, pages 182–201. Springer, 2021a. URL https://doi.org/10.1007/978-3-030-89051-3_11.

Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. Latent effects for reusable language components: Extended version. *CoRR*, abs/2108.11155, 2021b. URL https://arxiv.org/abs/2108.11155.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.*, 2(POPL): 8:1–8:30, 2018. URL https://doi.org/10.1145/3158096.

Richard S. Bird and Lambert G. L. T. Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction,* *MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998. URL https://doi.org/10.1007/BFb0054285.

Richard S. Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects Comput.*, 11(2):200–222, 1999. URL https://doi.org/10.1007/s001650050047.

Matthias Blume, Umut A. Acar, and Wonseok Chae. Extensible programming with first-class cases. In John H. Reppy and Julia L. Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 239–250. ACM, 2006. URL https://doi.org/10.1145/1159803.1159836.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA): 126:1–126:30, 2020. URL https://doi.org/10.1145/3428194.

Edwin C. Brady. Programming and reasoning with algebraic effects and dependent types. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 133–144, 2013a. URL https://doi.org/10.1145/2500365.2500581.

Edwin C. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013b. URL https://doi.org/10.1017/S095679681300018X.

Yufei Cai, Paolo G. Giarrusso, and Klaus Ostermann. System f-omega with equirecursive types for datatype-generic programming. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 30–43. ACM, 2016. doi: 10.1145/2837614.2837660. URL https://doi.org/10.1145/2837614.2837660.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009a. URL https://doi.org/10.1017/S0956796809007205.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009b. URL https://doi.org/10.1017/S0956796809007205.

Pietro Cenciarelli and Eugenio Moggi. A syntactic approach to modularity in denotational semantics. 1993.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 3–14. ACM, 2010a. ISBN 978-1-60558-794-3. URL https://doi.org/10.1145/1863543.1863547.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 3–14. ACM, 2010b. ISBN 978-1-60558-794-3. URL https://doi.org/10.1145/1863543.1863547.

James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. System F in agda, for fun and profit. In Graham Hutton, editor, *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, volume 11825 of *Lecture Notes in Computer Science*, pages 255–297. Springer, 2019. doi: 10.1007/978-3-030-33636-3_10. URL https://doi.org/10.1007/978-3-030-33636-3_10.

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. URL https://doi.org/10.1145/1411204.1411226.

Martin Churchill and Peter D. Mosses. Modular bisimulation theory for computations and values. In Frank Pfenning, editor, *FOSSACS 2013*, volume 7794 of *LNCS*, pages 97–112. Springer, 2013.

Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. Reusable components of semantic specifications. *LNCS Trans. Aspect Oriented Softw. Dev.*, 12:132–179, 2015.

Matteo Cimini, Dale Miller, and Jeremy G. Siek. Extrinsically typed operational semantics for functional languages. In Ralf Lämmel, Laurence Tratt, and Juan de Lara, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 108–125. ACM, 2020. URL https://doi.org/10.1145/3426425.3426936.

Koen Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9(3):313–323, 1999. URL https://doi.org/10.1017/s0956796899003342.

Jesper Cockx. Dependent pattern matching and proof-relevant unification. 2017. URL https://lirias.kuleuven.be/handle/123456789/583556.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Doo bee doo bee doo. *J. Funct. Program.*, 30:e9, 2020. URL https://doi.org/10.1017/S0956796820000039.

Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83. Citeseer, 1992.

Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December*

1988, Proceedings, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988. ISBN 3-540-52335-9. URL https://doi.org/10.1007/3-540-52335-9_47.

Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses - practical extensibility with object algebras. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2012. ISBN 978-3-642-31056-0. URL https://doi.org/10.1007/978-3-642-31057-7_2.

Bruno C. d. S. Oliveira, Shin-Cheng Mu, and Shu-Hung You. Modular reifiable matching: a list-of-functors approach to two-level types. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 82–93. ACM, 2015. URL https://doi.org/10.1145/2804302.2804315.

Pierre-Évariste Dagand. *A cosmology of datatypes : reusability and dependent types*. PhD thesis, University of Strathclyde, Glasgow, UK, 2013a. URL http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=22713.

Pierre-Évariste Dagand. *A cosmology of datatypes : reusability and dependent types*. PhD thesis, University of Strathclyde, Glasgow, UK, 2013b. URL http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=22713.

Pierre-Évariste Dagand. The essence of ornaments. *J. Funct. Program.*, 27:e9, 2017. URL https://doi.org/10.1017/S0956796816000356.

Pierre-Évariste Dagand and Conor McBride. Transporting functions across ornaments. *J. Funct. Program.*, 24(2-3):316–383, 2014. URL https://doi.org/10.1017/S0956796814000069.

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, 2013a. URL https://doi.org/10.1145/2429069.2429094.

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, 2013b. URL https://doi.org/10.1145/2429069.2429094.

Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C. d. S. Oliveira. Modular monadic meta-theory. pages 319–330, 2013c. URL https://doi.org/10.1145/2500365.2500587.

Marco Devesas Campos and Paul Blain Levy. A syntactic view of computational adequacy. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 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 10803 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 2018. ISBN 978-3-319-89365-5. URL https://doi.org/10.1007/978-3-319-89366-2_4.

Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. In *ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 143–155, 2011.

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992. URL https://doi.org/10.1016/0304-3975(92)90014-7.

Andrzej Filinski. Representing layered monads. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 175–188, 1999. URL https://doi.org/10.1145/292540.292557.

Marcelo P. Fiore and Sam Staton. Substitution, jumps, and algebraic effects. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 41:1–41:10, 2014. URL https://doi.org/10.1145/2603088.2603163.

Benedict R Gaster and Mark P Jones. A polymorphic type system for extensible records and variants. Technical report, Citeseer, 1996.

Johan Glimming and Neil Ghani. Difunctorial semantics of object calculus. In Viviana Bono, Michele Bugliesi, and Sophia Drossopoulou, editors, *Proceedings of the Second Workshop on Object Oriented Developments, WOOD 2004, London, UK, August 30, 2004*, volume 138 of *Electronic Notes in Theoretical Computer Science*, pages 79–94. Elsevier, 2004. URL https://doi.org/10.1016/j.entcs.2005.09.012.

Joseph A Goguen. An intial algebra approach to the specification, correctness and implementation of abstract data types. *IBM Research Report*, 6487, 1976.

Peter G. Hancock and Anton Setzer. Interactive programs in dependent type theory. In Peter Clote and Helmut Schwichtenberg, editors, *Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000, Proceedings*, volume 1862 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2000. URL https://doi.org/10.1007/3-540-44622-2_21.

Robert Harper. A simplified account of polymorphic references. *Inf. Process. Lett.*, 51(4):201–206, 1994.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. URL https://doi.org/10.1145/363235.363259.

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. URL https://doi.org/10.1007/BFb0053567.

Alex Hubers and J. Garrett Morris. Generic programming with extensible data types: Or, making ad hoc extensible data types less ad hoc. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023. URL https://doi.org/10.1145/3607843.

Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the programming language haskell, A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5):1, 1992. URL https://doi.org/10.1145/130697.130699.

Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3):70–99, 2006. URL https://doi.org/10.1016/j.tcs.2006.03.013.

Mauro Jaskelioff. Monatron: An extensible monad transformer library. In *Implementation and Application of Functional Languages - 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers*, pages 233–248, 2008. URL https://doi.org/10.1007/978-3-642-24452-0_13.

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*, 1995. Springer. ISBN 3-540-59451-5. URL https://doi.org/10.1007/3-540-59451-5.

Patricia Johann and Neil Ghani. Initial algebra semantics is enough! In Simona Ronchi Della Rocca, editor, *Typed Lambda*

Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings, volume 4583 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2007. URL https://doi.org/10.1007/978-3-540-73228-0_16.

Patricia Johann and Enrico Ghiorzi. Parametricity for nested types and gadts. *Log. Methods Comput. Sci.*, 17(4), 2021. URL https://doi.org/10.46298/lmcs-17(4:23)2021.

Patricia Johann and Andrew Polonsky. Higher-kinded data types: Syntax and semantics. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019. URL https://doi.org/10.1109/LICS.2019.8785657.

Patricia Johann, Enrico Ghiorzi, and Daniel Jeffries. Parametricity for primitive nested types. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12650 of *Lecture Notes in Computer Science*, pages 324–343. Springer, 2021. URL https://doi.org/10.1007/978-3-030-71995-1_17.

Thomas Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer, 1987. URL https://doi.org/10.1007/3-540-18317-5_10.

Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In Jeuring and Meijer [1995], pages 97–136. ISBN 3-540-59451-5. URL https://doi.org/10.1007/3-540-59451-5_4.

Mark P. Jones and Luc Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, New

Haven, Connecticut, USA, December 1993. URL http://web.cecs.pdx.edu/~mpj/pubs/RR-1004.pdf.

Guy L. Steele Jr. Building interpreters by composing monads. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 472–492. ACM Press, 1994. URL https://doi.org/10.1145/174675.178068.

Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158. ACM, 2013. URL https://doi.org/10.1145/2500365.2500590.

Steven Keuchel and Tom Schrijvers. Generic datatypes à la carte. In Jacques Carette and Jeremiah Willcock, editors, *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming, WGP 2013, Boston, Massachusetts, USA, September 28, 2013*, pages 13–24. ACM, 2013. URL https://doi.org/10.1145/2502488.2502491.

Richard B. Kieburtz and Jeffrey Lewis. Programming with algebras. In Jeuring and Meijer [1995], pages 267–307. ISBN 3-540-59451-5. URL https://doi.org/10.1007/3-540-59451-5_8.

Oleg Kiselyov. Typed tagless final interpreters. In Jeremy Gibbons, editor, *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, volume 7470 of *Lecture Notes in Computer Science*, pages 130–174. Springer, 2010. ISBN 978-3-642-32201-3. URL https://doi.org/10.1007/978-3-642-32202-0_3.

Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105. ACM, 2015. URL https://doi.org/10.1145/2804302.2804319.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In Chung-chieh Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 59–70. ACM, 2013. URL `https://doi.org/10.1145/2503778.2503791`.

Donald E. Knuth. *Literate programming*, volume 27 of *CSLI lecture notes series*. Center for the Study of Language and Information, 1992. ISBN 978-0-937073-81-0.

Hsiang-Shang Ko and Jeremy Gibbons. Programming with ornaments. *J. Funct. Program.*, 27:e2, 2017. URL `https://doi.org/10.1017/S0956796816000307`.

Saul A Kripke. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly*, 9 (5-6):67–96, 1963.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014. URL `https://doi.org/10.1145/2535838.2535841`.

Daan Leijen. Type directed compilation of row-typed algebraic effects. 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 486–499. ACM, 2017. URL `https://doi.org/10.1145/3009837.3009872`.

Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. URL `https://doi.org/10.1145/1538788.1538814`.

Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004. ISBN 1-4020-1730-8.

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, 1995a. URL https://doi.org/10.1145/199448.199528.

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, 1995b. ISBN 0-89791-692-1. URL https://doi.org/10.1145/199448.199528.

Sam Lindley and James Cheney. Row-based effect types for database integration. In Benjamin C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 91–102. ACM, 2012. URL https://doi.org/10.1145/2103786.2103798.

Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

Ken Madlener, Sjaak Smetsers, and Marko C. J. D. van Eekelen. Formal component-based semantics. In Michel A. Reniers and Pawel Sobocinski, editors, *Proceedings Eight Workshop on Structural Operational Semantics 2011, SOS 2011, Aachen, Germany, 5th September 2011*, volume 62 of *EPTCS*, pages 17–29, 2011. URL https://doi.org/10.4204/EPTCS.62.2.

Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984. ISBN 978-88-7088-228-5.

Conor McBride. Ornamental algebras, algebraic ornaments. Unpublished manuscript, 2011.

Lambert G. L. T. Meertens. Paramorphisms. *Formal Aspects Comput.*, 4(5):413–424, 1992. URL https://doi.org/10.1007/BF01211391.

Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991. URL https://doi.org/10.1007/3540543961_7.

Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. URL https://doi.org/10.1016/0022-0000(78)90014-4.

Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989. URL https://doi.org/10.1109/LICS.1989.39155.

Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, LFCS, University of Edinburgh, 1990. URL http://www.disi.unige.it/person/MoggiE/ftp/abs-view.ps.gz.

Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991. URL https://doi.org/10.1016/0890-5401(91)90052-4.

J. Garrett Morris. Variations on variants. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 71–81. ACM, 2015. URL https://doi.org/10.1145/2804302.2804320.

J. Garrett Morris and James McKinna. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.*, 3(POPL):12:1–12:28, 2019a. URL https://doi.org/10.1145/3290325.

J. Garrett Morris and James McKinna. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.*, 3(POPL):12:1–12:28, 2019b. URL https://doi.org/10.1145/3290325.

Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:195–228, 2004. URL https://doi.org/10.1016/j.jlap.2004.03.008.

Alan Mycroft, Dominic A. Orchard, and Tomas Petricek. Effect systems revisited - control-flow algebra and semantics. In Christian W. Probst, Chris Hankin, and René Rydhof Hansen, editors, *Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, volume 9560 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2016. ISBN 978-3-319-27809-4. URL https://doi.org/10.1007/978-3-319-27810-0_1.

Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer, 1999. URL https://doi.org/10.1007/3-540-48092-7_6.

Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008. ISBN 978-3-642-04651-3. URL https://doi.org/10.1007/978-3-642-04652-0_5.

Ulf Norell. Dependently typed programming in agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009. URL https://doi.org/10.1145/1481861.1481862.

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 589–615. Springer, 2016. URL https://doi.org/10.1007/978-3-662-49498-1_23.

Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. Towards improved GADT reasoning in scala. In Jonathan Immanuel Brachthäuser, Sukyoung Ryu, and Nathaniel Nystrom, editors, *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019*, pages 12–16. ACM, 2019. URL https://doi.org/10.1145/3337932.3338813.

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. URL https://doi.org/10.1145/53990.54010.

Benjamin C. Pierce. *Basic category theory for computer scientists*. Foundations of computing. MIT Press, 1991. ISBN 978-0-262-66071-6.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

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. URL https://doi.org/10.1016/j.entcs.2014.10.015.

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. Syntax and semantics for operations with scopes. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 809–818. ACM, 2018. URL https://doi.org/10.1145/3209108.3209166.

Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

Gordon D. Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. ISBN 3-540-43366-X. URL https://doi.org/10.1007/3-540-45931-6_24.

Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Appl. Categorical Struct.*, 11(1):69–94, 2003. URL https://doi.org/10.1023/A:1023064908962.

Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009a. URL https://doi.org/10.1007/978-3-642-00590-9_7.

Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009b. ISBN 978-3-642-00589-3. URL https://doi.org/10.1007/978-3-642-00590-9_7.

Matija Pretnar. Inferring algebraic effects. *Log. Methods Comput. Sci.*, 10(3), 2014. URL https://doi.org/10.2168/LMCS-10(3:21)2014.

Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. In Dan R. Ghica, editor, *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 19–35. Elsevier, 2015. URL https://doi.org/10.1016/j.entcs.2015.12.003.

Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 77–88. ACM Press, 1989. URL https://doi.org/10.1145/75277.75284.

Cas van der Rest and Casper Bach Poulsen. Towards a language for defining reusable programming language components - (project paper). In Wouter Swierstra and Nicolas Wu, editors, *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers*, volume 13401 of *Lecture Notes in Computer Science*, pages 18–38. Springer, 2022. URL https://doi.org/10.1007/978-3-031-21314-4_2.

Cas van der Rest and Casper Bach Poulsen. Types and semantics for extensible data types (extended version). *CoRR*, abs/2309.14985, 2023. URL https://doi.org/10.48550/arXiv.2309.14985.

Cas van der Rest and Casper Bach Poulsen. GitHub - heft-lang/hefty-equations: Modular reasoning about (elaborations of) higher-order effects — github.com. https://github.com/heft-lang/hefty-equations, 2024.

Cas van der Rest and Wouter Swierstra. A completely unique account of enumeration. *Proc. ACM Program. Lang.*, 6(ICFP): 411–437, 2022. URL https://doi.org/10.1145/3547636.

Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. Intrinsically-typed definitional interpreters à la carte (artifact), 2022a. URL https://doi.org/10.5281/zenodo.7074690.

Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter D. Mosses. Intrinsically-typed definitional interpreters à la carte. *Proc. ACM Program. Lang.*, 6(OOPSLA2): 1903–1932, 2022b. URL https://doi.org/10.1145/3563355.

van der Rest, Cas and Casper Bach Poulsen. Types and semantics for extensible data types. In Chung-Kil Hur, editor, *Programming Languages and Systems - 21st Asian Symposium, APLAS 2023, Taipei, Taiwan, November 26-29, 2023, Proceedings*, volume 14405 of *Lecture Notes in Computer Science*, pages 46–66. Springer, 2023. URL https://doi.org/10.1007/978-981-99-8311-7_3.

John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. URL https://doi.org/10.1023/A:1010027404223.

Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6): 397–434, 2010. URL https://doi.org/10.1016/j.jlap.2010.03.012.

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. 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 284–298. ACM, 2020. URL https://doi.org/10.1145/3372885.3373818.

Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. Intrinsically typed compilation with nameless labels. *Proc. ACM Program.*

Lang., 5(POPL):1–28, 2021. doi: 10.1145/3434303. URL https://doi.org/10.1145/3434303.

David Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.

Tom Schrijvers, Nicolas Wu, Benoit Desouter, and Bart Demoen. Heuristics entwined with handlers combined: From functional specification to logic programming implementation. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 259–270, 2014. URL https://doi.org/10.1145/2643135.2643145.

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: what binds them together. 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 98–113. ACM, 2019. URL https://doi.org/10.1145/3331545.3342595.

Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs in agda. In Matthew Might, David Van Horn, Andreas Abel, and Tim Sheard, editors, *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, pages 3–12. ACM, 2013. URL https://doi.org/10.1145/2428116.2428120.

Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. *Proceedings of the Symposium on Computers and Automata*, 21, 01 1971.

Neil Sculthorpe, Paolo Torrini, and Peter D. Mosses. A modular structural operational semantics for delimited continuations. In Olivier Danvy and Ugo de'Liguoro, editors, *Proceedings of the Workshop on Continuations, WoC 2016, London, UK, April 12th 2015*, volume 212 of *EPTCS*, pages 63–80, 2015. URL https://doi.org/10.4204/EPTCS.212.5.

Christopher Strachey. Towards a formal semantics. 1966.

Christopher S. Strachey. Fundamental concepts in programming languages. *High. Order Symb. Comput.*, 13(1/2):11–49, 2000. URL https://doi.org/10.1023/A:1010000313106.

Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008. URL https://doi.org/10.1017/S0956796808006758.

Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000. URL https://doi.org/10.1016/S0304-3975(00)00053-0.

Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997.

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. URL https://doi.org/10.4204/EPTCS.191.13.

Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nord. J. Comput.*, 6(3):343, 1999.

Phil Wadler. The expression problem. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, 1998. Accessed: 2020-07-01.

Philip Wadler. The essence of functional programming. In Ravi Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 1–14. ACM Press, 1992. URL https://doi.org/10.1145/143165.143169.

Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. July 2020. URL http://plfa.inf.ed.ac.uk/20.07/.

Mitchell Wand. Final algebra semantics and data type extensions. *J. Comput. Syst. Sci.*, 19(1):27–44, 1979. URL https://doi.org/10.1016/0022-0000(79)90011-4.

Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 92–97. IEEE Computer Society, 1989. URL https://doi.org/10.1109/LICS.1989.39162.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. URL https://doi.org/10.1006/inco.1994.1093.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 1–12. ACM, 2014. URL https://doi.org/10.1145/2633357.2633358.

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. URL https://doi.org/10.1145/3371119.

Zhixuan Yang and Nicolas Wu. Reasoning about effect interaction by fusion. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. URL https://doi.org/10.1145/3473578.

Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. Structured handling of scoped effects. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 462–491. Springer, 2022. URL https://doi.org/10.1007/978-3-030-99336-8_17.

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. URL https://doi.org/10.1145/3473572.

Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. Compositional programming. *ACM Trans. Program. Lang. Syst.*, 43(3): 9:1–9:61, 2021. URL https://doi.org/10.1145/3460228.

Yizhou Zhang and Andrew C. Myers. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.*, 3(POPL): 5:1–5:29, 2019. URL https://doi.org/10.1145/3290318.

PUBLICATIONS

- Casper Bach Poulsen, Cas van der Rest, and Tom Schrijvers. Staged effects and handlers for modular languages with abstraction. In *Workshop on Partial Evaluation and Program Manipulation (PEPM). http://casperbp. net/store/staged-effects-and-handlers. pdf*, 2021
- Cas van der Rest and Casper Bach Poulsen. Towards a language for defining reusable programming language components - (project paper). In Wouter Swierstra and Nicolas Wu, editors, *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers*, volume 13401 of *Lecture Notes in Computer Science*, pages 18–38. Springer, 2022. URL https://doi.org/10.1007/978-3-031-21314-4_2
- Cas van der Rest and Wouter Swierstra. A completely unique account of enumeration. *Proc. ACM Program. Lang.*, 6(ICFP):411–437, 2022. URL https://doi.org/10.1145/3547636
- Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter D. Mosses. Intrinsically-typed definitional interpreters à la carte. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1903–1932, 2022b. URL https://doi.org/10.1145/3563355
- Casper Bach Poulsen and Cas van der Rest. Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proc. ACM Program. Lang.*, 7(POPL):1801–1831, 2023. URL https://doi.org/10.1145/3571255

- van der Rest, Cas and Casper Bach Poulsen. Types and semantics for extensible data types. In Chung-Kil Hur, editor, *Programming Languages and Systems - 21st Asian Symposium, APLAS 2023, Taipei, Taiwan, November 26-29, 2023, Proceedings*, volume 14405 of *Lecture Notes in Computer Science*, pages 46–66. Springer, 2023. URL https://doi.org/10.1007/978-981-99-8311-7_3

SUMMARY

Type systems are a tool for preventing software errors, by classifying (sub)terms according to how they are evaluated. This way, mistakes can be caught at *compile-time*, ruling out the existence of entire classes of mistakes altogether. Using a programming language with a strong type system to develop critical software can dramatically reduce the prevalence and impact of bugs.

In light of the potentially enormous impact of bugs, it is important that we can trust a type system to be succesful in preventing errors. A key property of type systems that reflects this criterium is *type soundness*, which establishes that "well-typed programs cannot go wrong". That is, programs that are deemed safe by the type system should not exhibit certain wrong behaviour when executed. To gain trust in a type system's ability to prevent errors, we can give a formal system of both a language's type system and *semantics*, and mathematically verify that the type system is sound with respect to the defined semantics. While this provides airtight evidence that a type system is succesful in ruling out certain mistakes, the formal specification and verification of programming languages requires a formidable amount of time and expertise on behalf of the language designer, and is therefore infeasible in most cases.

This thesis aims to cut down on the cost associated with the formal specification and verification of programming languages, through reuse of (parts of) existing specifications and proofs. Specifically, we investigate how to build *reusable programming language components*. Underlying this research is the belief that the effort expended to specify and verify a new programming language should focus on the novel features of the language, and that we should benefit as much as possible from pre-existing verification efforts for other parts of the language. To realize this vision, it is necessary to make progress on two fronts. First, we must understand how the semantics of programming languages

can be formally specified and verified in a modular way. An important challenge in defining a language's semantics modularly is to give a modular description of its *side effects*. While *algebraic effects and handlers* provide a solution to this problem for many familiar side effects, *higher-order* effects cannot be described using the approach. Second, the *meta language* used to develop specifications should feature adequate support for reuse in order to eliminate syntactic overhead incurred by modularity as much as possible.

Part I of this thesis contributes several semantic techniques for defining *modular intrinsically-typed definitional interpreters* in a dependently-typed host language, such as Agda. More specifically, in Chapter 2 we explain how intrinsically-typed definitional interpreters can be defined modularly, such that the composition of interpreters preserves the encoded soundness invariant. Then, in Chapter 3, we explore how to define and reason about higher-order effects in a modular way. A modular treatment of higher-order effects is crucial for defining the semantics of many widely-used language constructs in a modular way, such as λ-abstraction or exception catching.

In part II of this thesis we investigate the design of *meta languages* for the purpose of defining reusable programming language components. In Chapter 4, we discuss the design of a language that features built-in support for modular algebraic data types and higher-order effects. This combination of features allows the definition of reusable language components, while incurring minimal syntactic overhead resulting from modularity of the definition. Chapter 5 puts this design on a stronger formal foundation, by developing a typing discipline for modular data types, together with a categorical semantics, illustrating how this feature can be incorporated in the design of meta-languages in a principled manner.

The contributions of this thesis addresses several important open challenges for developing reusable programming language components, such as how to preserve type safety invariants when composing intrinsically-typed interpreters, modular definitions and reasoning for higher-order effects, and how to integrate modularity in the design of functional meta languages. However, before reuse becomes a feasible approach for large-scale language projects, there is still work to be done. Most importantly, we still lack a uniform treatment of intrinsically-typed side effects, such as polymorphic references, and the support for modularity in meta languages would have to be extended to include intrinsically-typed definitional interpreters as well.

SAMENVATTING

Typesystemen zijn een middel om softwarefouten te voorkomen, door (sub)termen te classificeren aan de hand van hoe ze geëvalueerd worden. Op deze manier kunnen fouten gesignaleerd worden *tijdens het compileren*, waarmee bepaalde soorten fouten in het geheel uitgesloten kunnen worden. Door een programmeertaal te gebruiken die uitgerust is met een sterk typesysteem voor het ontwikkelen van kritieke software, kan de frequentie en invloed van bugs dramatisch gereduceerd worden.

In het licht van de potentieel enorme invloed van bugs is het belangrijk dat we erop kunnen vertrouwen dat een typesysteem er inderdaad in slaagt om fouten te voorkomen. Een sleuteleigenschap van typesystemen die dit criterium vangt is *typejuistheid*, wat vaststelt dat "correct getypeerde programma's niet fout kunnen gaan". Dat wil zeggen, programma's die als veilig beoordeeld worden door het typesysteem vertonen inderdaad bepaald verkeerd gedrag niet wanneer ze uitgevoerd worden. Het vertrouwen in het vermogen van een typesysteem om fouten te voorkomen kan verhoogd worden door een formele specificatie te geven van zowel het typesysteem alsmede de *semantiek* van de programmeertaal in kwestie, en vervolgens juistheid van het typesysteem ten opzichte van de gedefiniëerde semantiek wiskundig te verifiëren. Hoewel dit ons waterdicht bewijs verschaft dat een typesysteem bepaalde fouten uitsluit, is het in de meeste gevallen onuitvoerbaar gezien de uitzonderlijke hoeveelheid tijd en expertise die nodig is voor de formele specificatie en verificatie van programmeertalen.

Dit proefschrift heeft als doel de kosten van het formeel specificeren en verifiëren van programmeertalen te beperken middels hergebruik van (delen van) bestaande specificaties en bewijzen. Meer specifiek onderzoeken we hoe *herbruikbare programmeertaalonderdelen* gebouwd kunnen worden. Ten grondslag aan dit onderzoek ligt de overtuiging dat de inspanning die verricht wordt om een nieuwe programmeertaal te specificeren en verifiëren zich zou moeten richten op de vernieuwende onderdelen van de taal, en dat voor andere delen van de taal zoveel mogelijk geprofiteerd zou moeten worden van bestaande bewijsinspanningen. Om deze visie te realiseren is het noodzakelijk dat we progressie boeken op twee fronten. Ten eerste dienen we te begrijpen hoe de semantiek van programmeertalen op een modulaire manier gespecificeerd en geverifiëerd kan worden. Een belangrijke uitdaging als het aankomt op het modulair specificeren van de semantiek van een taal, is om haar *neveneffecten* op een modulaire manier te beschrijven. Hoewel *algebraïsche effecten en handlers* het modulariteitsprobleem oplossen voor veel mogelijke neveneffecten, kunnen *effecten van een hogere orde* niet met deze aanpak beschreven worden. Ten tweede zou de *meta-taal* die gebruikt wordt om specificaties in te ontwikkelen uitgerust moeten zijn met adequate ondersteuning voor hergebruik zodat de syntactische kosten ten gevolge van modulariteit tot een minimum beperkt worden.

In deel I van dit proefschrift dragen we verscheidene semantische technieken bij om modulaire *intrinsiek getypeerde definitie-interpretators* in een afhankelijk-getypeerde gasttaal, zoals Agda, te kunnen definiëren. Meer specifiek bespreken we in hoofdstuk 2 hoe intrinsiek getypeerde definitie-interpretators op modulaire manier gedefinieerd kunnen worden, zodanig dat het samenvoegen van interpretators hun typejuistheid behoudt. Vervolgens onderzoeken we in hoofdstuk 3 het op modulaire wijze beschrijven van en redeneren over effecten van een hogere orde. Het vermogen om effecten van een hogere orde op een modulaire manier te kunnen beschrijven is cruciaal voor het definiëren van veelvoorkomende taalonderdelen, zoals λ-abstractie of functionaliteit voor het afvangen van excepties.

In deel II van dit proefschrift onderzoeken we het ontwerp van *meta-talen* die als doel hebben om herbruikbare programmeertaalonderdelen te beschrijven. In hoofdstuk 4 bespreken we het ontwerp van een taal die ingebouwde ondersteuning voor modulaire algebraische datatypes en effecten van een hogere orde bevat. Deze combinatie van functionaliteiten stelt ons in staat om herbruikbare programmeertaalonderdelen te definiëren terwijl de syntactische kosten als gevolg van modulariteit van de definities minimaal blijven. Hoofdstuk 5 verschaft dit taalontwerp van een sterkere formele fundering door een typesysteem te ontwikkelen voor modulaire datatypes in combinatie met een categorische semantiek, waarmee we illustreren hoe dergelijke functionaliteit op een principiële manier opgenomen kan worden in het ontwerp van een meta-taal.

De bijdragen in dit proefschrift geven antwoord op een aantal belangrijke open uitdagingen voor het ontwikkelen van herbruikbare programmeertaalonderdelen, zoals het behouden van de typejuistheidsinvariant wanneer intrinsiek getypeerde definitie-interpretators worden samengevoegd, het modulair definiëren en redeneren over effecten van een hogere orde, en hoe modulariteit in het ontwerp van functionele meta-talen geïntegreerd kan worden. Desondanks is er voldoende werk te verzetten voordat hergebruik een uitvoerbare strategie kan worden voor grootschalige taalprojecten. In het bijzonder missen we nog een uniforme beschrijving van intrinsiek getypeerde neveneffecten, zoals polymorfe referenties, en de ondersteuning van modulariteit in meta-talen zal moeten worden uitgebreid naar intrinsiek getypeerde definitie-interpretators.

CURRICULUM VITAE

1995

Born on June 1st in Rotterdam, The Netherlands.

2007 - 2013

High School (VWO/gymnasium), Marnix Gymnasium, Rotterdam, The Netherlands.

2013 - 2017

Bachelor Computer Science, Utrecht University, Utrecht, The Netherlands

2017 - 2019

Master Computer Science, Utrecht University, Utrecht, The Netherlands. Thesis title: "Generating Constrained Test Data Using Datatype Generic Programming", supervised by dr. Wouter Swierstra. Graduated with *cum laude* distinction.

2019 - 2024

Doctoral candidate in Computer Science, Delft University of Technology, Delft, The Netherlands. Thesis title: "Reusable Programming Language Components", supervised by dr. Casper Bach Poulsen.

2024 - present

Formal Methods Engineer at Input Output Global.