THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Modular Normalization with Types

Nachiappan Valliappan

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2023

Modular Normalization with Types

Nachiappan Valliappan

ISBN 978-91-7905-850-0
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 5316
ISSN 0346-718X

Department of Computer Science & Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31-772 1000

Printed by Chalmers Reproservice,
Gothenburg, Sweden, 2023

Modular Normalization with Types
Nachiappan Valliappan
Department of Computer Science & Engineering
Chalmers University of Technology

Abstract

With the increasing use of software in today's digital world, software is becoming more and more complex and the cost of developing and maintaining software has skyrocketed. It has become pressing to develop software using effective tools that reduce this cost. Programming language research aims to develop such tools using mathematically rigorous foundations. A recurring and central concept in programming language research is *normalization*: the process of transforming a complex expression in a language to a canonical form while preserving its meaning. Normalization has compelling benefits in theory and practice, but is extremely difficult to achieve. Several program transformations that are used to optimise programs, prove properties of languages and check program equivalence, for instance, are after all instances of normalization, but they are seldom viewed as such.

Viewed through the lens of current methods, normalization lacks the ability to be broken into sub-problems and solved independently, i.e., lacks *modularity*. To make matters worse, such methods rely excessively on the syntax of the language, making the resulting normalization algorithms brittle and sensitive to changes in the syntax. When the syntax of the language evolves due to modification or extension, as it almost always does in practice, the normalization algorithm may need to be revisited entirely. To circumvent these problems, normalization is currently either abandoned entirely or concrete instances of normalization are achieved using ad hoc means specific to a particular language. Continuing this trend in programming language research poses the risk of building on a weak foundation where languages either lack fundamental properties that follow from normalization or several concrete instances end up being repeated in an ad hoc manner that lacks reusability.

This thesis advocates for the use of type-directed *Normalization by Evaluation* (NbE) to develop normalization algorithms. NbE is a technique that provides an opportunity for a modular implementation of normalization algorithms by allowing us to disentangle the syntax of a language from its semantics. Types further this opportunity by allowing us to dissect a language into isolated fragments, such as functions and products, with an individual specification of syntax and semantics. To illustrate type-directed NbE in context, we develop NbE algorithms and show their applicability for typed programming language calculi in three different domains (modal types, static information-flow control and categorical combinators) and for a family of embedded-domain specific languages in Haskell.

Keywords: programming language theory, normalization, type systems

Acknowledgments

My research at Chalmers has been a constant tug of war between what I would like to do and what needs to be done. Should I follow the elegant path of theoretical pursuit? Or must I solve today's problems and push the boundaries of current technology? These questions, in addition to my wide range of interests, have often left me conflicted and occasionally in a state of despair. I cannot imagine advising me being an easy job, and yet Alejandro Russo has been patient and supportive at all the times when it mattered the most. I am grateful for his advice and guidance.

This thesis began with Andreas Abel handing me his notes on Normalization by Evaluation (NbE), the topic of this thesis, from his desk while declaring "these have been waiting for you". I am deeply grateful to Andreas for suggesting NbE and for all the recommendations and technical advice he has offered me over the years.

I am very fortunate to have had Carlos Tomé Cortiñas and Fabian Ruch as my peers, collaborators and friends over the course of developing this thesis. They have helped me refine my intuition-driven approach and I have benefited immensely from the countless hours we have spent together discussing technical and philosophical matters. Fabian has been an unofficial technical advisor since the inception of this thesis, and I am thankful for all his suggestions and enthusiasm towards NbE.

The programming languages community is a wonderful lot of passionate people and I am glad to be a part of it. Dominic Orchard reached out to me at a critical moment when I was doubtful about completing this thesis, and I am grateful for his advice in the discussion that followed. Sam Lindley and Ohad Kammar have taken a considerable interest in both the development of this thesis and my career as a researcher. Sandro Stucki, Thierry Coquand, Graham Leigh and several anonymous reviewers have offered valuable comments and suggestions on several parts of this thesis. I am indebted to all of them for their advice, criticism and encouragement.

Without the unwavering care and support of my parents, I would not be in a position of luxury where I dedicate my time only to things I am interested in. My friends and colleagues at Chalmers—including Matthí, Agustín, Abhiroop, Irene, Jeremy, Robert, Ivan, Mohammad, Elisabet, Alejandro, and Benjamin—have made this journey an enjoyable experience. Without the early encouragement of Murali Krishnan and Richa George, none of this would have happened, and without the enjoyable experiences with my friends, none of this would be worth it. Thank you!

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023).

Nachi. May 5, 2023.

Contents

Overview

Introduction

The thesis underlying this bundle of papers, henceforth called a *thesis*, is:

> *Type-directed Normalization by Evaluation is a solution to the problem of developing modular normalization algorithms that are robust to extension.*

Normalization is a broad term used for the process of transforming a program into a canonical shape while preserving its meaning. The objective of normalization can be fundamental (e.g., checking program equivalence or proving a property of a program) or more practical (e.g., optimizing performance or analyzing a program). By "modular" normalization, I mean the ability to decompose a normalization algorithm into independent modules that can be reused under different circumstances.

Normalization algorithms are currently implemented by rewriting the syntax of a given program in accordance with certain *reduction* rules. For example, the following reduction rule specifies that an expression $0 + x$ can be rewritten to x.

$$0 + x \;\mapsto\; x$$

With sufficient reduction rules and sophisticated rewriting strategies, normalization can be achieved even for complex languages. Rewriting techniques are neither the enemy nor an ally of this thesis, but the difficulty with normalizing by rewriting syntax lies in its very nature: it is a process sensitive to the syntax of the language. When the syntax of the language is modified or extended, a rewriting algorithm may need to be revisited entirely. The goal of this thesis is to develop modular normalization algorithms that are robust to modification and extension.

To achieve its goal, this thesis[1] employs a normalization technique known as *Normalization by Evaluation* (NbE) in combination with the *types* of a language. NbE avoids rewriting and instead normalizes a program by evaluating it in a suitable semantic domain. NbE provides an opportunity for a modular implementation of normalization by decoupling the syntax of a language from its semantics. Types further this opportunity by allowing us to dissect a language into isolated fragments, such as functions and products, with an individual specification of syntax and semantics. Explaining the sorcery of NbE and illustrating its potential in the presence of types for implementing modular normalization algorithms for well-typed functional programming languages is the main non-technical contribution of this thesis.

[1] an extended version of my licentiate thesis [38]

I.1 Why Normalization Matters

In the design and implementation of programming languages, normalization is a recurring concept of central importance. The main benefit of normalization lies in its ability to reduce infinitely large equivalence classes of terms identified by their semantics to their normal forms, thus vastly reducing the set of terms that we must take into consideration while reasoning about the language. In programming languages, normalization may have several objectives:

- *Checking program equivalence*: How do we know if the integer expressions $2 + 2 * (x - 1)$ and $4 * (x - 1)$ are equal? We can normalize them to $2 * x$ and $(4 * x) - 4$ respectively, and observe that they are not equal unless $x = 2$. Normalization is widely used to check the equivalence of programs and proofs in the implementation of dependently typed languages and proof assistants.

- *Implementing program optimization*: Normalization can be used to optimise a program. The integer expression $2 + 2 * (x - 1)$ contains the unnecessary overhead of evaluating known arithmetic operations on literal numbers, and can be optimally replaced by $2 * x$ without changing its meaning.

- *Proving properties of complex type systems*: Type systems enable the detection and prevention of errors in a program before executing it by associating every expression in the language with a type. For example, the type assignment `2 : Int` denotes that the expression literal `2` has the integer type `Int`. The integrity of a complex type system lies within its ability to correctly associate a value to its expected type, and not, for example, incorrectly associate a string literal `"hello"` to the type `Int`. This property, called *canonicity*, can be proved with normalization by showing that canonical forms of all (closed) expressions with type `Int` are in fact integers.

- *Proving completeness of semantic specification*: How do we know that a specification of the semantics of a language is complete? That is, how do we know that we have not missed an intended equivalence between two expressions in a language such as $(x + y) * z \approx (x * z) + (y * z)$? Normalization allows us to prove completeness with respect to a semantic model of the language that defines the expected specification, and thus allows us to gain confidence in its completeness with respect to the model.

Normalization is also prevalent in other areas of logic and computer science. For example, in formal logic, normalization is used to prove meta-theoretic properties of a proof system such as logical consistency and the subformula property. In formal verification, normalization is used to convert a logical formula to a normal form for the purpose of deciding its truth. Similarly normalization is also used in databases to eliminate data redundancy and improve the integrity of data in a database. This thesis is developed in the context of programming languages, particularly well-typed functional programming languages, but it may have applications beyond this area.

I.2 The Sorcery of Normalization by Evaluation

This subsection gives an introduction to the essence of NbE by illustrating the implementation of an NbE algorithm for an extremely simple language: arithmetic expressions consisting of the addition of natural numbers. For this purpose, we encode the natural numbers 0 as *Zero*, 1 as *Succ Zero*, 2 as *Succ* (*Succ Zero*), and so on, using a constant symbol *Zero* and a successor function *Succ*. Addition in a given expression can be reduced using one of the two following reduction steps.

$$\begin{aligned} \mathit{Zero} + x &\mapsto x \\ (\mathit{Succ}\ x) + y &\mapsto x + (\mathit{Succ}\ y) \end{aligned}$$

The reduction relation $\mapsto$ specifies how an expression must be reduced to another expression. Using this specification, the expression $1 + 2$ can be normalized to 3 by reducing it as follows.

$$\begin{aligned} &\mathit{Succ}\ \mathit{Zero} + \mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}) && (1 + 2) \\ &\quad\mapsto \mathit{Zero} + \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})) && (0 + 3) \\ &\quad\mapsto \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})) && (3) \end{aligned}$$

Observe that we rewrite the expression twice before reaching the normal form, which cannot be reduced anymore since none of the reduction steps apply. Complex expressions may need to be rewritten several times before a normal form is reached. Rewriting is the basis for traditional normalization procedures, while NbE, on the other hand, does not involve any rewriting.

NbE achieves normalization in two steps: 1) *evaluating* the expressions in a "host" language, and 2) *quoting* (sometimes called *reifying*) the resulting values back to expressions. Let us implement NbE for our example language using the programming language Haskell as the host.

- *Evaluation:* We implement the first step using an interpreter function called `eval`. This function interprets natural numbers as integers and the addition of natural numbers by addition of integers.

```
eval :: Expr Nat -> Int
eval Zero     = 0
eval (Succ x) = eval x + 1
eval (x + y)  = eval x + eval y
```

- *Quotation:* The second step is to invert the integer values back to natural number expressions, and is implemented by a function called `quote`. This function need not be defined on all integer values, but only on the values that may be returned by `eval`.

```
quote :: Int -> Expr Nat
quote 0 = Zero
quote n = Succ (quote (n - 1))
```

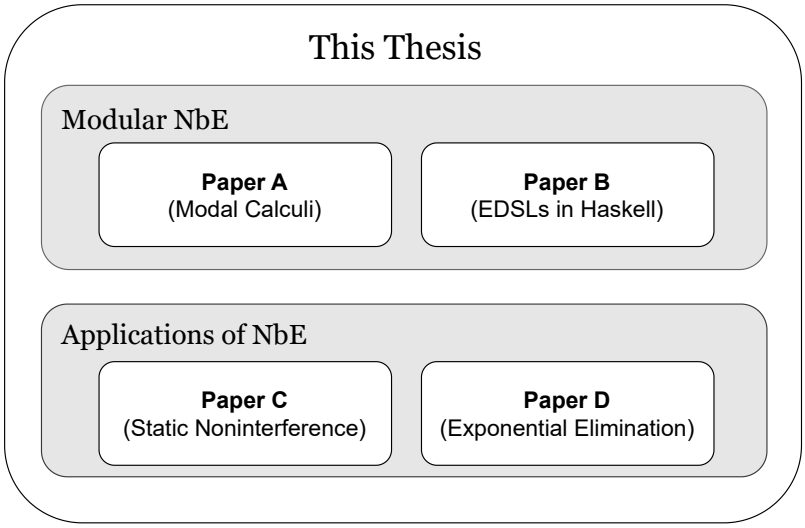


Figure I.1: Outline of this thesis

We implement the normalization procedure by a function `norm` that applies `quote` on the result of `eval`.

```
norm :: Expr Nat -> Expr Nat
norm e = quote (eval e)
```

Observe that an invocation of `norm` on the expression *Succ Zero* + *Succ* (*Succ Zero*) does indeed return its normal form *Succ* (*Succ* (*Succ Zero*)). `norm` uses the ability of Haskell to evaluate the addition of integers to normalize the addition of natural numbers. This function can be extended easily to other arithmetic operators, and, with some care, even to support variables and other unknowns in expressions.

This seemingly simple idea to leverage a host language's evaluation mechanism to normalize expressions extends much beyond arithmetic expressions, and has found a wide range of applications. NbE has been used to achieve normalization results in various programming calculi [2, 7, 9, 18, 24, 30], decide equality in algebraic structures [4], typecheck dependently-typed programming languages [3, 25], and to prove completeness [5, 17] and coherence [10] theorems. NbE algorithms have been observed to yield much faster normalization than their rewriting counterparts [8, 29], and there is also evidence that indicates that it can be used to speed up compilation in optimizing compilers [29].

I.3 Fistful of Problems and This Thesis

This section gives an overview of the problems addressed in this thesis by the papers in the forthcoming chapters. These problems occur independently in different

domains, and thus the following subsections may be read in any order. These subsections discuss the interest in these problems (and their domains) and provide an introduction to the corresponding chapters—see Figure I.1 for an outline.

I.3.1 Fitch-Style Modal Calculi

Modal types In type systems, a *modality* can be broadly construed as a unary type constructor with certain properties. Type systems with modalities have found a wide range of applications in programming languages to capture and specify properties of a program in its type. For example, in language-based security, a field dedicated to developing secure programming languages, a substantial number of languages use modalities to ensure sensitive data is not leaked to an unauthorized principal [27]. Using a modal type `Secret Int`, the programmer can indicate via the modality `Secret` to the type system that the underlying integer value must be kept a secret. The type system automatically tracks the flow of this integer in the program and prevents the need for a careful and error-prone manual analysis. Modal type systems offer a form of lightweight and low cost alternative to formal verification of programs for preventing software errors since type systems are a familiar abstraction used widely in mainstream programming languages.

The design and implementation of modal type systems for various applications is a vibrant area of ongoing research. Different applications may demand different modal operations, which means there can be several different kinds of modalities. The *necessity* modality is one such modality that has found applications in modelling purity in an impure functional language [13], confidentiality in information-flow control [32], and binding-time separation in partial evaluation and staged computation [20].

Fitch-style modal calculi Fitch-style modal lambda calculi [12, 16, 31] feature necessity modalities in a typed lambda calculus by extending the typing context with a delimiting "lock" operator. The characteristic lock operator simplifies formulating calculi that incorporate different modal operations and these calculi have excellent computational properties. Each variant demands, however, different, tedious and seemingly ad hoc treatment to prove meta-theoretic properties such as normalization. In Chapter A, we identify the *possible-world* semantics of Fitch-style calculi and use it to develop normalization. The possible-world semantics enables a modular implementation of normalization for various Fitch-style calculi by isolating their differences to a specific parameter that identifies the modal fragment. We showcase several consequences of normalization for proving meta-theoretic properties of Fitch-style calculi based on different interpretations of the necessity modality in programming languages, such as capability safety, noninterference and a form of binding-time correctness.

I.3.2 Embedded Domain-Specific Languages

Overview An *embedded* domain-specific language (eDSL) is an implementation of a domain-specific language (DSL) as a library in a host language. Implementing a DSL as an eDSL offers two main advantages:

- The programmer can leverage the features of the host, typically a more powerful general purpose programming language, to write programs in the eDSL.
- Developing an eDSL compiler requires much lesser effort than building a dedicated DSL compiler, since the host language's compiler can be reused for standard compilation phases such as lexical analysis, parsing and type-checking.

The tradeoff, however, is that programming an eDSL may require some familiarity of the host language.

Let us consider the example with arithmetic expressions again. The following library functions in Haskell constitute an eDSL to write simple arithmetic expressions.

```
val :: Int -> Expr Int
(+) :: Expr Int -> Expr Int -> Expr Int
(*) :: Expr Int -> Expr Int -> Expr Int
```

Using these functions, we can write the expression `1+2` as `val 1 + val 2`.

Suppose that we would like to write an expression x^n that represents the n-th power of an expression x, for some known non-negative integer n. How should we do this when exponentiation is not a primitive function provided by the eDSL? If this were a mere DSL, we would write $x * x * x$ for x^3, for example, since multiplication is provided. In an eDSL, however, we can take this a step further to write a generic `power` function that generates this expression automatically for an arbitrary integer n.

```
power :: Int -> Expr Int -> Expr Int
power n x = if (n <= 0) then x else (x * (power (n - 1))
```

Using the `power` function, we may write `power 8 x` for x^8 instead of $x * x * x * x * x * x * x * x$. The former variant is concise, less error-prone and also makes it easy to modify and reuse code.

Notice that the definition of the power function uses Haskell's features such as conditionals (`if ...`), comparison (`n <= 0`) and function recursion (`power (n - 1)`). Even if the eDSL does not implement these features natively, we are able to use them to write expressions. EDSLs make it easy to derive additional functionality by leveraging those the host language. EDSLs, specifically in Haskell, have found a wide range of applications: hardware description [11], digital signal-processing [6], runtime verification [21, 35], parallel and distributed programming [14, 23], GPU programming [15]—and the list goes on.

Compiling EDSLs In an eDSL program we may think of a value of type `Int` as a *static* integer that is known at compile-time, and a value of type `Expr Int` as a *dynamic* integer that is known only at runtime. This *stage separation* of values as static and dynamic corresponds to a manual form of *binding-time analysis* in partial evaluation [26], and presents an opportunity to exploit Haskell's execution mechanism to evaluate static computations in an eDSL program.

Though separation of stages enables the programmer to manually specify those parts of an eDSL program that must be evaluated by Haskell, it also burdens them to maintain multiple variants of the same program. In addition to `power` function defined above, we may also desire the several variants of the exponentiation function as follows, each corresponding to a different separation of stages for its arguments and result.

```
power0 :: Int -> Int -> Int
power1 :: Int -> Expr Int -> Expr Int
power2 :: Expr Int -> Int -> Expr Int
power3 :: Int -> Int -> Expr Int
...
```

NbE offers a modular solution to this problem by making specialization automatic, without the need for manual stage separation. Chapter B shows that typed NbE is particularly well-suited for specializing eDSL programs in Haskell given the natural reliance on a host language. We argue that existing techniques for embedding DSLs in Haskell (e.g., [37]), which may at first seem somewhat ad hoc, can be viewed as instances of NbE after all.

I.3.3 Language-Based Security

Information-Flow Control. Information-Flow Control (IFC) is a language-based security enforcement technique that guarantees the confidentiality of sensitive data by controlling how information is allowed to flow in a program. The guarantee that programs secured by an IFC system do not leak sensitive data is often proved using a property called noninterference. Noninterference ensures that an observer authorized to view the output of a program (pessimistically called the attacker) cannot infer any sensitive data handled by the program from its output.

Proof by Normalization. To prove that an IFC system ensures noninterference, we must show that the public output of secured programs remain unaffected by variations in its secret inputs. If the output remains unaffected by a given input, then it must be the case that it does not depend on the input to compute the output—thus ensuring that the attacker could not possibly learn about the secret inputs. Such programs may *refer* to the secret input in its body, but they must not *use* it to compute the public output.

Chapter C proposes a new syntax-directed proof strategy to prove noninterference for well-typed programming calculi that enforce static IFC. The key idea of this chapter is to use normalization to eliminate any unnecessary input references in a program, leaving behind references that are only absolutely necessary to compute the result. Noninterference is then proved by ensuring that no public output depends on a reference to a secret input in the normal form of a program—a task that is much simpler than most semantics-based proof techniques. This technique is illustrated for a model of the terminating fragment of the seclib library [36] in Haskell, which is a simply-typed lambda calculus extended with IFC primitives.

I.3.4 Categorical Combinators

Combinator Calculi. Combinators can be understood as program building blocks which can be assembled in various ways to construct programs. In functional programming, a combinator is a primitive higher order function, which can be applied to and composed with other combinators to build more complex functions. Unlike programming languages based on the lambda calculus, combinators lack a notion of variables. In practice, this means that programming using combinators can be an unbearable task and should probably be avoided at all costs. But then, why care about combinators at all?

> *"...roughly λ-calculus is well-suited for programming, and combinators (of Curry, or those introduced here) allow for implementations getting rid of some difficulties in the scope of variables."*
> —P.-L. Curien (1985, Typed Categorical Combinatory Logic)

The output of a function in the lambda calculus is computed using a process known as β-reduction. The primary difficulty with β-reduction lies in its very definition: the output of a function $\lambda x.b$ for some input i is computed by *substituting* all occurrences of the argument variable x, in the body of the function b, with the actual input i. This statement is succinctly captured by the β-rule:

$$(\lambda x.b)i \;\mapsto\; b[i/x]$$

This rule states that a function $\lambda x.b$ when applied to an argument i, can be reduced to a simpler term $b[i/x]$, which is the result of substituting all occurrences of x with i in the body of the function b. Although substitution readily appeals to the intuition of replacement, there are a number of auxiliary conditions that must be checked before the actual replacement of x with i. For this reason, substitution has long had a reputation for being notoriously difficult to implement and reason about.

Combinators, on the other hand, avoid the need for substitution by disallowing variables entirely. Instead, they adopt a style of reduction that relies on simply "shifting symbols". The (categorical) combinator equivalent of the β-rule is, what I like to call, the *exponential elimination* rule:

$$\mathit{apply} \circ \langle \Lambda b, i \rangle \;\mapsto\; b \circ \langle \mathit{id}, i \rangle$$

This rule reads as: the application (*apply*) of a function (Λb) to an argument (i) can be reduced to a composition of the body (b) with its input (i) in an appropriate manner. The operator $\_ \circ \_$ denotes the sequencing, or composition, of two combinators and $\langle \_, \_ \rangle$ denotes the coupling, or pairing, of two combinators. We shall return to the specifics of this rule in a later chapter, but simply observe here that it does not use the substitution operation on the right-hand side, and that the body of the function (b) remains unmodified.

The absence of substitution, an external operation, means that we need not impose additional correctness criteria over the computation rules—which is great news for formal reasoning! In essence, the very characteristic of combinators that makes them impractical for programming also makes them amenable to implementation and reasoning: the lack of variables.

Categorical Combinators. Categorical combinators are combinators designed after arrows, or *morphisms*, in category theory. They were introduced by Pierre-Louis Curien as an alternative to the SKI combinator calculus to implement functional programming languages.

The primary motivation behind categorical combinators appears to be two-fold: 1) to faithfully simulate reduction in lambda calculus without the difficulty of variable bindings, and 2) to establish a syntactic equivalence theorem between the lambda calculus and the categorical model underlying the combinators—namely, the *(free) cartesian closed categories*. Categorical combinators offered an appealing alternative to Church's more popular SKI combinator calculi, since their design is based on a semantic model. This means that the reduction rules of the combinators arise naturally from the model rather than having to be imposed.

> *"...categorical combinatory logic is entirely faithful to βreduction where [Curry's SKI] combinatory logic needs additional rather complex and unnatural axioms to be..."*
> —P.-L. Curien (1986, Categorical Combinators)

Categorical combinators were used to formulate the *Catergorical Abstract Machine* (CAM) [19], which was used to used to implement early versions of Caml—the predecessor of the OCaml programming language. Later versions of Caml, however, did not use CAM due to performance issues and difficulty with optimizations [2]. Despite its failure in use for compiling a programming language in practice, the ease of formulating an abstract machine for categorical combinators (noted in [1]) seems to have influenced several variants of CAM, an example of which is the Linear Abstract Machine [28].

In recent times, variants of (what appear to be) categorical combinators have reappeared in practical applications. They have been used to compile Haskell code using user-defined interpretations [22] and in the development of a language for executing smart contracts on the blockchain [34].

Exponential elimination. Exponentials are the equivalent of higher-order functions in categorical combinator calculi. The runtime representation of an exponential is a *closure*, a value accompanied by an environment. Adding support for closures complicates the implementation of the abstract machine, and makes certain static analyses difficult [39]. In [22], exponentials narrow the domain of target interpretations that are supported by the compiler.

The exponential elimination rule from earlier indicates that exponentials can be eliminated in a specific case. This makes us wonder: can exponentials be eliminated statically by applying this rule repetitively on a program? This would solve both the above problems. Without a careful analysis, however, it is difficult to answer this question, since there may be interactions with other rules in the calculus that prevent exponential elimination rule from being applied.

Chapter D shows that exponential elimination can be achieved for categorical combinators with sums and products, in the presence of a special *distributivity* combinator that distributes products over sums. The ability to erase the equivalent of

[2] https://caml.inria.fr/about/history.en.html

higher-order functions in functional calculus (known as *defunctionalization*) is not news [33], but the distributivity requirement is a somewhat surprising insight. A technical challenge faced by this result is the presence of the empty and sum types, both of which are known for making normalization notoriously difficult.

II

Statement of contributions

This thesis is a bundle of articles published at different venues focused on programming language research. The initial conception, overall development and writing of all these articles were led by me. This chapter outlines my individual contributions to their technical development alongside a listing of their abstracts.

A Normalization for Fitch-Style Modal Calculi

Nachiappan Valliappan, Fabian Ruch, Carlos Tomé Cortiñas

Fitch-style modal lambda calculi enable programming with necessity modalities in a typed lambda calculus by extending the typing context with a delimiting operator that is denoted by a lock. The addition of locks simplifies the formulation of typing rules for calculi that incorporate different modal axioms, but each variant demands different, tedious and seemingly ad hoc syntactic lemmas to prove normalization. In this work, we take a semantic approach to normalization, called normalization by evaluation (NbE), by leveraging the possible-world semantics of Fitch-style calculi to yield a more modular approach to normalization. We show that NbE models can be constructed for calculi that incorporate the K, T and 4 axioms of modal logic, as suitable instantiations of the possible-world semantics. In addition to existing results that handle β-equivalence, our normalization result also considers η-equivalence for these calculi. Our key results have been mechanized in the proof assistant Agda. Finally, we showcase several consequences of normalization for proving meta-theoretic properties of Fitch-style calculi as well as programming-language applications based on different interpretations of the necessity modality.

Statement of contributions I independently mechanized the first Agda prototype using the categorical semantics of Fitch-style calculi and identified the common pattern in the construction of their NbE models. Fabian showed me a connection to possible-world semantics in modal logic that gave a systematic and elegant account of this pattern. This convinced me to factor the construction of the NbE models through possible-world semantics, roughly midway during this development, from which point onwards Fabian and I co-developed the remaining technical results. Carlos helped us explore and understand the applications of these calculi.

Appeared in: *Proceedings of the ACM on Programming Languages Vol 6. ICFP (2022)*

B Practical Normalization by Evaluation for EDSLs

Nachiappan Valliappan, Alejandro Russo, Sam Lindley

Embedded domain-specific languages (eDSLs) are typically implemented in a rich host language, such as Haskell, using a combination of deep and shallow embedding techniques. While such a combination enables programmers to exploit the execution mechanism of Haskell to build and specialize eDSL programs, it blurs the distinction between the host language and the eDSL. As a consequence, extension with features such as sums and effects requires a significant amount of ingenuity from the eDSL designer. In this paper, we demonstrate that Normalization by Evaluation (NbE) provides a principled framework for building, extending, and customizing eDSLs. We present a comprehensive treatment of NbE for deeply embedded eDSLs in Haskell that involves a rich set of features such as sums, arrays, exceptions and state, while addressing practical concerns about normalization such as code expansion and the addition of domain-specific features.

Statement of contributions I developed all the technical results in this paper under the supervision of Alejandro. Sam helped us understand and survey earlier work (some unpublished) that set out to leverage NbE to embed DSLs.

Appeared in: *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell (2021)*

C Simple Noninterference by Normalization

Carlos Tomé Cortiñas, Nachiappan Valliappan

Information-flow control (IFC) languages ensure programs preserve the confidentiality of sensitive data. *Noninterference*, the desired security property of such languages, states that public outputs of programs must not depend on sensitive inputs. In this paper, we show that noninterference can be proved using normalization. Unlike arbitrary terms, normal forms of programs are well-principled and obey useful syntactic properties—hence enabling a simpler proof of noninterference. Since our proof is syntax-directed, it offers an appealing alternative to traditional semantic based techniques to prove noninterference.

In particular, we prove noninterference for a static IFC calculus, based on Haskell's `seclib` library, using normalization. Our proof follows by straightforward induction on the structure of normal forms. We implement normalization using *normalization by evaluation* and prove that the generated normal forms preserve semantics. Our results have been verified in the Agda proof assistant.

Statement of contributions Carlos and I shared the technical development in this work. I constructed most of the NbE model and proved it correct, while Carlos helped me understand, formulate and prove noninterference.

Appeared in: *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (2019)*

D Exponential Elimination for Bicartesian Closed Categorical Combinators

Nachiappan Valliappan, Alejandro Russo

Categorical combinators offer a simpler alternative to typed lambda calculi for static analysis and implementation. Since categorical combinators are accompanied by a rich set of conversion rules which arise from categorical laws, they also offer a plethora of opportunities for program optimization. It is unclear, however, how such rules can be applied in a systematic manner to eliminate intermediate values such as *exponentials*, the categorical equivalent of higher-order functions, from a program built using combinators. Exponential elimination simplifies static analysis and enables a simple closure-free implementation of categorical combinators—reasons for which it has been sought after.

In this paper, we prove exponential elimination for *bicartesian closed* categorical (BCC) combinators using normalization. We achieve this by showing that BCC terms can be normalized to normal forms which obey a weak subformula property. We implement normalization using Normalization by Evaluation, and also show that the generated normal forms are correct using logical relations.

Statement of contributions I developed all the technical results in this paper under the supervision of Alejandro.

Appeared in: *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (2019)*

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of functional programming*, 1(4):375–416, 1991.

[2] A. Abel and C. Sattler. Normalization by evaluation for call-by-push-value and polarized lambda calculus. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*, pages 1–12, 2019.

[3] A. Abel and H. Talk. Normalization by evaluation: Dependent types and impredicativity. *Unpublished. http://www.tcs.ifi.lmu.de/~abel/habil.pdf*, 2013.

[4] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310. IEEE, 2001.

[5] T. Altenkirch and T. Uustalu. Normalization by evaluation for $\lambda\!\rightarrow 2$. In *International Symposium on Functional and Logic Programming*, pages 260–275. Springer, 2004.

[6] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 169–178. IEEE, 2010.

[7] V. Balat, R. Di Cosmo, and M. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. *ACM SIGPLAN Notices*, 39(1):64–76, 2004.

[8] U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation. In *Prospects for Hardware Foundations*, pages 117–137. Springer, 1998.

[9] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. 1991.

[10] I. Beylin and P. Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In *International Workshop on Types for Proofs and Programs*, pages 47–61. Springer, 1995.

[11] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in haskell. *ACM SIGPLAN Notices*, 34(1):174–184, 1998.

[12] V. Borghuis. *Coming to terms with modal logic : on the interpretation of modalities in typed lambda-calculus*. PhD thesis, Mathematics and Computer Science, 1994.

[13] V. Choudhury and N. Krishnaswami. Recovering purity with comonads and capabilities. *Proc. ACM Program. Lang.*, 4(ICFP):111:1–111:28, 2020.

[14] K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.

[15] K. Claessen, M. Sheeran, and J. Svensson. Obsidian: Gpu programming in haskell. *Designing Correct Circuits*, page 101, 2008.

[16] R. Clouston. Fitch-style modal lambda calculi. In C. Baier and U. D. 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 258–275. Springer, 2018.

[17] C. Coquand. From semantics to rules: A machine assisted analysis. In *International Workshop on Computer Science Logic*, pages 91–105. Springer, 1993.

[18] T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1):75–94, 1997.

[19] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of computer programming*, 8(2):173–202, 1987.

[20] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[21] F. Dedden. Compiling an haskell edsl to c: A new c back-end for the copilot runtime verification framework. Master's thesis, 2018.

[22] C. Elliott. Compiling to categories. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–27, 2017.

[23] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, pages 118–129, 2011.

[24] A. Filinski. Normalization by evaluation for the computational lambda-calculus. In *International Conference on Typed Lambda Calculi and Applications*, pages 151–165. Springer, 2001.

[25] D. Gratzer, J. Sterling, and L. Birkedal. Implementing a modal dependent type theory. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

[26] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.

[27] G. A. Kavvos. Modalities, cohesion, and information flow. *Proc. ACM Program. Lang.*, 3(POPL):20:1–20:29, 2019.

[28] Y. Lafont. The linear abstract machine. *Theor. Comput. Sci.*, 59:157–180, 1988.

[29] S. Lindley. Normalisation by evaluation in the compilation of typed functional programming languages. 2005.

[30] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

[31] S. Martini and A. Masini. A computational interpretation of modal proofs. In *Proof theory of modal logic (Hamburg, 1993)*, volume 2 of *Appl. Log. Ser.*, pages 213–241. Kluwer Acad. Publ., Dordrecht, 1996.

[32] K. Miyamoto and A. Igarashi. A modal foundation for secure information flow. In *Workshop on Foundations of Computer Security*, pages 187–203, 2004.

[33] S. Najd, S. Lindley, J. Svenningsson, and P. Wadler. Everything old is new again: quoted domain-specific languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 25–36, 2016.

[34] R. O'Connor. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 107–120, 2017.

[35] L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: a hard real-time runtime monitor. In *International Conference on Runtime Verification*, pages 345–359. Springer, 2010.

[36] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. *ACM Sigplan Notices*, 44(2):13–24, 2008.

[37] J. Svenningsson and E. Axelsson. Combining deep and shallow embedding of domain-specific languages. *Comput. Lang. Syst. Struct.*, 44:143–165, 2015.

[38] N. Valliappan. *Be My Guest: Normalizing and Compiling Programs Using a Host Language*. PhD thesis, Chalmers Tekniska Hogskola (Sweden), 2020.

[39] N. Valliappan, S. Mirliaz, E. L. Vesga, and A. Russo. Towards adding variety to simplicity. In *International Symposium on Leveraging Applications of Formal Methods*, pages 414–431. Springer, 2018.

Papers

A

Normalization for Fitch-Style Modal Calculi

Abstract. Fitch-style modal lambda calculi enable programming with necessity modalities in a typed lambda calculus by extending the typing context with a delimiting operator that is denoted by a lock. The addition of locks simplifies the formulation of typing rules for calculi that incorporate different modal axioms, but each variant demands different, tedious and seemingly ad hoc syntactic lemmas to prove normalization. In this work, we take a semantic approach to normalization, called normalization by evaluation (NbE), by leveraging the possible-world semantics of Fitch-style calculi to yield a more modular approach to normalization. We show that NbE models can be constructed for calculi that incorporate the K, T and 4 axioms of modal logic, as suitable instantiations of the possible-world semantics. In addition to existing results that handle β-equivalence, our normalization result also considers η-equivalence for these calculi. Our key results have been mechanized in the proof assistant Agda. Finally, we showcase several consequences of normalization for proving metatheoretic properties of Fitch-style calculi as well as programming-language applications based on different interpretations of the necessity modality.

1 INTRODUCTION

In type systems, a *modality* can be broadly construed as a unary type constructor with certain properties. Type systems with modalities have found a wide range of applications in programming languages to specify properties of a program in its type. In this work, we study typed lambda calculi equipped with a *necessity* modality (denoted by $\Box$) formulated in the so-called Fitch style.

The necessity modality originates from modal logic, where the most basic intuitionistic modal logic IK (for "intuitionistic" and "Kripke") extends intuitionistic propositional logic with a unary connective $\Box$, the *necessitation rule* (if $\cdot \vdash A$ then $\Gamma \vdash \Box A$) and the *K axiom* ($\Box(A \Rightarrow B) \Rightarrow \Box A \Rightarrow \Box B$). With the addition of further modal axioms T ($\Box A \Rightarrow A$) and 4 ($\Box A \Rightarrow \Box\Box A$) to IK, we obtain richer logics IT (adding axiom T), IK4 (adding axiom 4), and IS4 (adding both T and 4). Type systems with necessity modalities based on IK and IS4 have found applications in partial evaluation and staged computation [14, 15], information-flow control [31], and recovering purity in an effectful language [11]. While type systems based on IT and IK4 do not seem to have any prior known programming applications, they are nevertheless interesting as objects of study that extend IK towards IS4.

Fitch-style modal lambda calculi [9, 12, 29] feature necessity modalities in a typed lambda calculus by extending the typing context with a delimiting "lock" operator (denoted by 🔒). In this paper, we consider the family of Fitch-style modal lambda calculi that correspond to the logics IK, IT, IK4, and IS4. These calculi extend the simply-typed lambda calculus (STLC) with a type constructor $\Box$, along with introduction and elimination rules for $\Box$ types formulated using the 🔒 operator. For instance, the calculus λ_{IK}, which corresponds to the logic IK, extends STLC with Rules $\Box$-Intro and $\lambda_{\text{IK}}/\Box$-Elim, as summarized in Fig. 1. The rules for λ-abstraction and function application are formulated in the usual way—but note the modified variable rule Var!

$$\textit{Ty} \qquad A ::= \ldots \mid \Box A \qquad\qquad \textit{Ctx} \qquad \Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, 🔒$$

$$\textsc{Var}\ \frac{}{\Gamma, x : A, \Gamma' \vdash x : A}\ 🔒 \notin \Gamma' \qquad \Box\textsc{-Intro}\ \frac{\Gamma, 🔒 \vdash t : A}{\Gamma \vdash \text{box}\, t : \Box A} \qquad \lambda_{\text{IK}}/\Box\textsc{-Elim}\ \frac{\Gamma \vdash t : \Box A}{\Gamma, 🔒, \Gamma' \vdash \text{unbox}_{\lambda_{\text{IK}}}\, t : A}\ 🔒 \notin \Gamma'$$

Fig. 1. Typing rules for λ_{IK} (omitting λ-abstraction and application)

The equivalence of terms in STLC is extended by Fitch-style calculi with the following rules for $\Box$ types, where the former states the β- (or computational) equivalence, and the latter states a type-directed η- (or extensional) equivalence.

$$\Box\text{-}\beta \quad \text{unbox}\,(\text{box}\, t) \sim t \qquad\qquad \Box\text{-}\eta \quad \frac{\Gamma \vdash t : \Box A}{t \sim \text{box}\,(\text{unbox}\, t)}$$

We are interested in the problem of normalizing terms with respect to these equivalences. Traditionally, terms in a calculus are normalized by rewriting them using rewrite rules formulated from these equivalences, and a term is said to be in *normal form* when it cannot be rewritten further. For example, we may formulate a rewrite

rule $\mathsf{unbox}\,(\mathsf{box}\,t) \mapsto t$ by orienting the $\Box$-β equivalence from left to right. This naive approach to formulating a rewrite rule, however, is insufficient for the $\Box$-η rule since normalizing with a rewrite rule $t \mapsto \mathsf{box}\,(\mathsf{unbox}\,t)$ (for $\Gamma \vdash t : \Box A$) does not terminate as it can be applied infinitely many times. It is presumably for this reason that existing normalization results [12] for some of these calculi only consider β-equivalence.

While it may be possible to carefully formulate a more complex set of rewrite rules that take the context of application into consideration to guarantee termination (as done, for example, by Jay and Ghani [25] for function and product types), the situation is further complicated for Fitch-style calculi by the fact that we must repeat such syntactic rewriting arguments separately for each calculus under consideration. The calculi λ_{IT}, λ_{IK4}, and λ_{IS4} differ from λ_{IK} only in the $\Box$-elimination rule, as summarized in Fig. 2. In spite of having identical syntax and term equivalences, each calculus

$$\lambda_{\mathrm{IT}}/\Box\text{-Elim}\quad \frac{\Gamma \vdash t : \Box A}{\Gamma, \Gamma' \vdash \mathsf{unbox}_{\lambda_{\mathrm{IT}}}\, t : A}\ \#_{🔒}(\Gamma') \leq 1 \qquad \lambda_{\mathrm{IK4}}/\Box\text{-Elim}\quad \frac{\Gamma \vdash t : \Box A}{\Gamma, 🔒, \Gamma' \vdash \mathsf{unbox}_{\lambda_{\mathrm{IK4}}}\, t : A}$$

$$\lambda_{\mathrm{IS4}}/\Box\text{-Elim}\quad \frac{\Gamma \vdash t : \Box A}{\Gamma, \Gamma' \vdash \mathsf{unbox}_{\lambda_{\mathrm{IS4}}}\, t : A}$$

Fig. 2. $\Box$-elimination rules for λ_{IT}, λ_{IK4}, and λ_{IS4}

demands different, tedious and seemingly ad hoc syntactic renaming lemmas [12, Lemmas 4.1 and 5.1] to prove normalization.

In this paper, we take a semantic approach to normalization, called normalization by evaluation (NbE) [8]. NbE bypasses rewriting entirely, and instead normalizes terms by evaluating them in a suitable semantic model and then reifying values in the model as normal forms. For Fitch-style calculi, NbE can be developed by leveraging their possible-world semantics. To this end, we identify the parameters of the possible-world semantics for the calculi under consideration, and show that NbE models can be constructed by instantiating those parameters. The NbE approach exploits the semantic overlap of the Fitch-style calculi in the possible-world semantics and isolates their differences to a specific parameter that determines the modal fragment, thus enabling the reuse of the evaluation machinery and many lemmas proved in the process.

In Section 2, we begin by providing a brief overview of the main idea underlying this paper. We discuss the uniform interpretation of types for four Fitch-style calculi (λ_{IK}, λ_{IT}, λ_{IK4} and λ_{IS4}) in possible-world models and outline how NbE models can be constructed as instances. The *reification* mechanism that enables NbE is performed alike for all four calculi. In Section 3, we construct an NbE model for λ_{IK} that yields a correct normalization algorithm, and then show how NbE models can also be constructed for λ_{IS4}, and for λ_{IT} and λ_{IK4} by slightly varying the instantiation. The

calculi λ_{IK} and λ_{IS4} and their normalization algorithms have been implemented and verified correct [41] in the proof assistant Agda [3].

NbE models and proofs of normalization in general have several useful consequences for term calculi. In Section 4, we show how NbE models and the accompanying normalization algorithm can be used to prove meta-theoretic properties of Fitch-style calculi including completeness, decidability, and some standard results in modal logic in a *constructive* manner. In Section 5, we discuss applications of our development to specific interpretations of the necessity modality in programming languages, and show (but do not mechanize) how application-specific properties that typically require semantic intervention can be proved syntactically. We show that properties similar to capability safety, noninterference, and binding-time correctness can be proved syntactically using normal forms of terms.

2 MAIN IDEA

The main idea underlying this paper is that normalization can be achieved in a modular fashion for Fitch-style calculi by constructing NbE models as instances of their possible-world semantics. In this section, we observe that Fitch-style calculi can be interpreted in the possible-world semantics for intuitionistic modal logic with a minor refinement that accommodates the 🔒 operator, and give a brief overview of how we construct NbE models as instances.

Possible-World Semantics. The possible-world semantics for intuitionistic modal logic [10] is parameterized by a *frame* F and a *valuation* V_ι. A frame F is a triple (W, R_i, R_m) that consists of a type W of *worlds* along with two binary *accessibility* relations R_i (for "intuitionistic") and R_m (for "modal") on worlds that are required to satisfy certain conditions. An element $w : W$ can be thought of as a representation of the "knowledge state" about some "possible world" at a certain point in time. Then, $w\ R_i\ w'$ represents an increase in knowledge from w to w', and $w\ R_m\ v$ represents a possible passage from w to v. A valuation V_ι, on the other hand, is a family of types $V_{\iota,w}$ indexed by $w : W$ along with functions $wk_{\iota,w,w'} : V_{\iota,w} \to V_{\iota,w'}$ whenever $w\ R_i\ w'$. An element $p : V_{\iota,w}$ can be thought of as "evidence" for (the knowledge of) the truth of the *atomic* proposition ι at the world w. The requirement for functions $wk_{\iota,w,w'}$ enforces that the knowledge of the truth of ι at w is preserved as time moves on to w', and is neither forgotten nor contradicted by any new evidence learned at w'. There are no such requirements on a valuation V_ι with respect to the modal accessibility relation R_m.

Given a frame (W, R_i, R_m) and a valuation V_ι, we interpret (object) types A in *any* Fitch-style calculus as families of (meta) types $[\![A]\!]_w$ indexed by worlds $w : W$, following the work by Fischer-Servi [18], Ewald [16], Plotkin and Stirling [35], and Simpson [39] as below:

$$
\begin{aligned}
[\![\iota \quad\ ]\!]_w &= V_{\iota,w} \\
[\![A \Rightarrow B]\!]_w &= \forall w'.\, w\ R_i\ w' \to [\![A]\!]_{w'} \to [\![B]\!]_{w'} \\
[\![\Box A \quad]\!]_w &= \forall w'.\, w\ R_i\ w' \to \forall v.\, w'\ R_m\ v \to [\![A]\!]_v
\end{aligned}
$$

The nonmodal type formers are interpreted as in the Kripke semantics for intuitionistic propositional logic: the base type ι is interpreted using the valuation V_ι, and

function types $A \Rightarrow B$ at $w : W$ are interpreted as *families* of functions $[\![A]\!]_{w'} \to [\![B]\!]_{w'}$ indexed by $w' : W$ such that $w\ R_i\ w'$. Recall that the generalization to families is necessary for the interpretation of function types to be sound.

As for the interpretation of modal types, at $w : W$ the types $\Box A$ are interpreted by families of elements $[\![A]\!]_v$ indexed by those $v : W$ that are accessible from w via some $w' : W$ such that $w\ R_i\ w'$ and $w'\ R_m\ v$. In other words, $\Box A$ is true at a world w if A is necessarily true in "the future", whichever concrete possibility this may turn out to be. We remark that the interpretation of $\Box A$ as $\forall v.\, w\ R_m\ v \to [\![A]\!]_v$, as in classical modal logic without the first quantifier $\forall w'.\, w\ R_i\ w'$, requires additional conditions [10, 39] on frames that (some of) the NbE models we construct do not satisfy.

In order to extend the possible-world semantics of intuitionistic modal logic to Fitch-style calculi, we must also provide an interpretation of contexts and the 🔒 operator, which is unique to the Fitch style, in particular:

$$\begin{aligned}
[\![\cdot \quad]\!]_w &= \top \\
[\![\Gamma, A]\!]_w &= [\![\Gamma]\!]_w \times [\![A]\!]_w \\
[\![\Gamma, 🔒]\!]_w &= \textstyle\sum_u [\![\Gamma]\!]_u \times u\ R_m\ w
\end{aligned}$$

The empty context $\cdot$ and the context extension Γ, A of a context Γ with a type A are interpreted as in the Kripke semantics for STLC by the terminal family and the Cartesian product of the families $[\![\Gamma]\!]$ and $[\![A]\!]$, respectively. While the interpretation of types $\Box A$ can be understood as a statement about the future, the interpretation of contexts $\Gamma, 🔒$ can be understood as a dual statement about the past: $\Gamma, 🔒$ is true at a world w if Γ is true at *some* world u for which w is a possibility, i.e. $u\ R_m\ w$.

With the interpretation of contexts Γ and types A as (W, R_i)-indexed families $[\![\Gamma]\!]$ and $[\![A]\!]$ at hand, the interpretation of terms $t : \Gamma \vdash A$, also known as *evaluation*, in a possible-world model is given by a function $[\![-]\!] : \Gamma \vdash A \to (\forall w.\, [\![\Gamma]\!]_w \to [\![A]\!]_w)$ as follows. Clouston [12] shows that the interpretation of STLC in Cartesian closed categories (CCCs) extends to an interpretation of Fitch-style calculi in any CCC equipped with an adjunction by interpreting $\Box$ and 🔒 by the right and left adjoint as well as box and unbox using the right and left adjuncts, respectively. The key idea here is that, correspondingly, the interpretation of terms in the nonmodal fragment of Fitch-style calculi using the familiar CCC structure on (W, R_i)-indexed families extends to the modal fragment: the interpretation of $\Box$ in a possible-world model has a left adjoint that is denoted by our interpretation of 🔒. In summary, the possible-world interpretation of Fitch-style calculi can be given by instantiation of Clouston's *generic* interpretation in CCCs equipped with an adjunction.

Constructing NbE Models as Instances. To construct an NbE model for Fitch-style calculi, we must construct a possible-world model with a function $quote : (\forall w.\, [\![\Gamma]\!]_w \to [\![A]\!]_w) \to \Gamma \vdash_{\mathrm{NF}} A$ that inverts the denotation $(\forall w.\, [\![\Gamma]\!]_w \to [\![A]\!]_w)$ of a term to a derivation $\Gamma \vdash_{\mathrm{NF}} A$ in normal form. The normal forms for the modal fragment of λ_{IK} are defined below, where $\Gamma \vdash_{\mathrm{NE}} A$ denotes a special case of normal forms known as

neutral elements.

$$\textsc{Nf}/\Box\textsc{-Intro}\ \frac{\Gamma, \locked \vdash_{\text{NF}} t : A}{\Gamma \vdash_{\text{NF}} \mathsf{box}\, t : \Box A} \qquad \lambda_{\text{IK}}/\textsc{Ne}/\Box\textsc{-Elim}\ \frac{\Gamma \vdash_{\text{NE}} t : \Box A}{\Gamma, \locked, \Gamma' \vdash_{\text{NE}} \mathsf{unbox}_{\lambda_{\text{IK}}}\, t : A}\ \locked \notin \Gamma'$$

The normal forms for λ_{IT}, λ_{IK4}, and λ_{IS4} are defined similarly by varying the elimination rule as in their term typing rules in Fig. 2.

Following the work on NbE for STLC with possible-world[1] models [13], we instantiate the parameters that define possible-world models for Fitch-style calculi as follows: we pick contexts for W, *order-preserving embeddings* (sometimes called "weakenings", defined in the next section) $\Gamma \leq \Gamma'$ for $\Gamma\ R_i\ \Gamma'$, and neutral derivations $\Gamma \vdash_{\text{NE}} \iota$ as the valuation $V_{\iota,\Gamma}$. It remains for us to instantiate the parameter R_m and show that this model supports the *quote* function.

The instantiation of the modal parameter R_m in the possible-world semantics varies for each calculus and captures the difference between them. Recall that the syntaxes of the four calculi only differ in their elimination rules for $\Box$ types. When viewed through the lens of the possible-world semantics, this difference can be generalized as follows:

$$\Box\textsc{-Elim}\ \frac{\Delta \vdash t : \Box A}{\Gamma \vdash \mathsf{unbox}\, t : A}\ (\Delta \lhd \Gamma)$$

We generalize the relationship between the context in the premise and the context in the conclusion using a generic modal accessibility relation $\lhd$ between contexts. When viewed as a candidate for instantiating the R_m relation, this rule states that if $\Box A$ is derivable in some past world Δ, then we may derive A in the current world Γ. The various $\Box$-elimination rules for Fitch-style calculi can be viewed as instances of this generalized rule, where we define $\lhd$ in accordance with $\Box$-elimination rule of the calculus under consideration. For example, for λ_{IK}, we observe that the context of the premise in Rule $\lambda_{\text{IK}}/\Box$-Elim is Γ and that of the conclusion is $\Gamma, \locked, \Gamma'$ such that $\locked \notin \Gamma'$, and thus define $\Delta \lhd_{\lambda_{\text{IK}}} \Gamma$ as $\exists \Delta'.\, \locked \notin \Delta' \wedge \Gamma = \Delta, \locked, \Delta'$. Similarly, we define $\Delta \lhd_{\lambda_{\text{IS4}}} \Gamma$ as $\exists \Delta'.\, \Gamma = \Delta, \Delta'$ for λ_{IS4}, and follow this recipe for λ_{IT} and λ_{IK4}. Accordingly, we instantiate the R_m parameter in the NbE model with the corresponding definition of $\lhd$ in the calculus under consideration.

A key component of implementing the *quote* function in NbE models is *reification*, which is implemented by a family of functions $\mathit{reify}_A : \forall \Gamma.\, [\![A]\!]_\Gamma \to \Gamma \vdash_{\text{NF}} A$ indexed by a type A. While its implementation for the simply-typed fragment follows the standard, for the modal fragment we are required to give an implementation of $\mathit{reify}_{\Box A} : \forall \Gamma.\, [\![\Box A]\!]_\Gamma \to \Gamma \vdash_{\text{NF}} \Box A$. To reify a value of $[\![\Box A]\!]_\Gamma$, we first observe that $[\![\Box A]\!]_\Gamma = \forall \Gamma'.\, \Gamma \leq \Gamma' \to \forall \Delta.\, \Gamma' \lhd \Delta \to [\![A]\!]_\Delta$ by definition of $[\![-]\!]$ and the instantiations of R_i with $\leq$ and R_m with $\lhd$. By picking Γ for Γ' and $\Gamma, \locked$ for Δ, we get $[\![A]\!]_{\Gamma,\locked}$ since $\leq$ is reflexive and it can be shown that $\Gamma \lhd \Gamma, \locked$ holds for the calculi under consideration. By reifying the value $[\![A]\!]_{\Gamma,\locked}$ recursively, we get a normal form $\Gamma, \locked \vdash_{\text{NF}} n : A$, which can be used to construct the desired normal form $\Gamma \vdash_{\text{NF}} \mathsf{box}\, n : \Box A$ using the rule Nf/□-Intro.

[1] also called "Kripke" or "Kripke-style"

3 POSSIBLE-WORLD SEMANTICS AND NbE

In this section, we elaborate on the previous section by defining possible-world models and showing that Fitch-style calculi can be interpreted soundly in these models. Following this, we outline the details of constructing NbE models as instances. We begin with the calculus λ_{IK}, and then show how the same results can be achieved for the other calculi.

Before discussing a concrete calculus, we present some of their commonalities.

Types, Contexts and Order-Preserving Embeddings. The grammar of types and typing contexts for Fitch-style is the following.

$$\textit{Ty} \quad A ::= \iota \mid A \Rightarrow B \mid \Box A \qquad\qquad \textit{Ctx} \quad \Gamma ::= \cdot \mid \Gamma, A \mid \Gamma, 🔒$$

Types are generated by an uninterpreted base type ι, function types $A \Rightarrow B$, and modal types $\Box A$, and typing contexts are "snoc" lists of types and locks.

We define the relation of *order-preserving embeddings* (OPE) on typing contexts in Fig. 3. An OPE $\Gamma \leq \Gamma'$ embeds the context Γ into another context Γ' while preserving the order of types and the order and number of locks in Γ.

$$\mathsf{base} : \cdot \leq \cdot \qquad \frac{o : \Gamma \leq \Gamma'}{\mathsf{drop}\, o : \Gamma \leq \Gamma', A} \qquad \frac{o : \Gamma \leq \Gamma'}{\mathsf{keep}\, o : \Gamma, A \leq \Gamma', A}$$

$$\frac{o : \Gamma \leq \Gamma'}{\mathsf{keep}_{🔒}\, o : \Gamma, 🔒 \leq \Gamma', 🔒}$$

Fig. 3. Order-preserving embeddings

3.1 The Calculus λ_{IK}

3.1.1 Terms, Substitutions and Equational Theory. To define the intrinsically-typed syntax and equational theory of λ_{IK}, we first define a modal accessibility relation on contexts $\Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$, which expresses that context Γ extends $\Delta, 🔒$ to the right without adding locks. Note that $\Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$ exactly when $\exists \Delta'. 🔒 \notin \Delta' \wedge \Gamma = \Delta, 🔒, \Delta'$.

$$\mathsf{nil} : \Gamma \lhd_{\lambda_{\mathrm{IK}}} \Gamma, 🔒 \qquad\qquad \frac{e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma}{\mathsf{var}\, e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma, A}$$

Fig. 4. Modal accessibility relation on contexts (λ_{IK})

Fig. 5 presents the intrinsically-typed syntax of λ_{IK}. We will use both $\Gamma \vdash t : A$ and $t : \Gamma \vdash A$ to say that t denotes an (intrinsically-typed) term of type A in context Γ, and similarly for substitutions, which will be defined below. Instead of named variables as in Fig. 1, variables are defined using De Bruijn indices in a separate judgement $\Gamma \vdash_{\mathrm{VAR}} A$. The introduction and elimination rules for function types are like those in STLC, and the introduction rule for the type $\Box A$ is similar to that of Fig. 1. The elimination

$$\textsc{Var-Zero}\ \ \Gamma, A \vdash_{\mathrm{VAR}} \mathsf{zero} : A \qquad \textsc{Var-Succ}\ \ \frac{\Gamma \vdash_{\mathrm{VAR}} v : A}{\Gamma, B \vdash_{\mathrm{VAR}} \mathsf{succ}\, v : A} \qquad \textsc{Var}\ \ \frac{\Gamma \vdash_{\mathrm{VAR}} v : A}{\Gamma \vdash \mathsf{var}\, v : A} \qquad \Rightarrow\textsc{-Intro}\ \ \frac{\Gamma, A \vdash t : B}{\Gamma \vdash \lambda\, t : A \Rightarrow B}$$

$$\Rightarrow\textsc{-Elim}\ \ \frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{app}\, t\, u : B} \qquad \Box\textsc{-Intro}\ \ \frac{\Gamma, \locksymbol \vdash t : A}{\Gamma \vdash \mathsf{box}\, t : \Box A} \qquad \lambda_{\mathrm{IK}}/\Box\textsc{-Elim}\ \ \frac{\Delta \vdash t : \Box A \qquad e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma}{\Gamma \vdash \mathsf{unbox}_{\lambda_{\mathrm{IK}}}\, t\, e : A}$$

Fig. 5. Intrinsically-typed terms of λ_{IK}

rule $\lambda_{\mathrm{IK}}/\Box$-Elim is defined using the modal accessibility relation $\Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$ which relates the contexts in the premise and the conclusion, respectively. This relation replaces the side condition ($\locksymbol \notin \Gamma'$) in Fig. 1 and other $\Box$-elimination rules in Sections 1 and 2. Note that formulating the rule for the term $\mathsf{unbox}_{\lambda_{\mathrm{IK}}}$ with $e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$ as a second premise is in sharp contrast to Clouston [12, Fig. 1] where the relation is not mentioned in the term but formulated as the *side condition* $\Gamma = \Delta, \locksymbol, \Gamma'$ for some lock-free Γ'.

A term $\Gamma \vdash t : A$ can be *weakened*, which is a special case of *renaming*, with an OPE (see Fig. 3) using a function $wk : \Gamma \leq \Gamma' \to \Gamma \vdash A \to \Gamma' \vdash A$. Given an OPE $o : \Gamma \leq \Gamma'$, renaming the term using wk yields a term $\Gamma' \vdash wk\, o\, t : A$ in the weaker context Γ'. The unit element for wk is the identity OPE $\mathsf{id}_{\leq} : \Gamma \leq \Gamma$, i.e. $wk\, \mathsf{id}_{\leq}\, t = t$. Renaming arises naturally when evaluating terms and in specifying the equational theory (e.g. in the η rule of function type).

$$\Gamma \vdash_{\mathrm{s}} \mathsf{empty} : \cdot \qquad \frac{\Gamma \vdash_{\mathrm{s}} s : \Delta \qquad \Gamma \vdash t : A}{\Gamma \vdash_{\mathrm{s}} \mathsf{ext}\, s\, t : \Delta, A} \qquad \frac{\Theta \vdash_{\mathrm{s}} s : \Delta \qquad e : \Theta \lhd_{\lambda_{\mathrm{IK}}} \Gamma}{\Gamma \vdash_{\mathrm{s}} \mathsf{ext}_{\locksymbol}\, s\, e : \Delta, \locksymbol}$$

Fig. 6. Substitutions for λ_{IK}

Substitutions for λ_{IK} are inductively defined in Fig. 6. A judgement $\Gamma \vdash_{\mathrm{s}} s : \Delta$ denotes a substitution for a context Δ in the context Γ. Applying a substitution to a term $\Delta \vdash t : A$, i.e. $subst\, s\, t : \Gamma \vdash A$, yields a term in the context Γ. The substitution $\mathsf{id}_{\mathrm{s}} : \Gamma \vdash_{\mathrm{s}} \Gamma$ denotes the identity substitution, which exists for all Γ. As usual, it can be shown that terms are closed under the application of a substitution, and that it preserves the identity, i.e. $subst\, \mathsf{id}_{\mathrm{s}}\, t = t$. Substitutions are also closed under renaming and this operation preserves the identity as well.

The equational theory for λ_{IK}, omitting congruence rules, is specified in Fig. 7. As discussed earlier, λ_{IK} extends the usual rules in STLC (Rules $\Rightarrow$-β and $\Rightarrow$-η) with rules for the $\Box$ type (Rules $\Box$-β and $\Box$-η). The function $factor : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma \to \Delta, \locksymbol \leq \Gamma$, in Rule $\Box$-β, maps an element of the modal accessibility relation $e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$ to an OPE $\Delta, \locksymbol \leq \Gamma$. This is possible because the context Γ does not have any lock to the right of $\Delta, \locksymbol$.

$$\Rightarrow\text{-}\beta\quad \frac{\Gamma, A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{app}\ (\lambda\, t)\ u \sim \mathit{subst}\ (\mathsf{ext}\ \mathsf{id}_{\mathsf{s}}\ u)\ t}$$

$$\Rightarrow\text{-}\eta\quad \frac{\Gamma \vdash t : A \Rightarrow B}{\Gamma \vdash t \sim \lambda\ (\mathsf{app}\ (\mathit{wk}\ (\mathsf{drop}\ \mathsf{id}_{\leq})\ t)\ (\mathsf{var}\ \mathsf{zero}))}$$

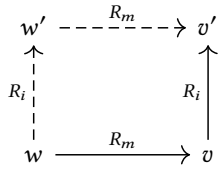
$$\Box\text{-}\beta\quad \frac{\Delta, \blacksquare \vdash t : A \qquad e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma}{\Gamma \vdash \mathsf{unbox}_{\lambda_{\mathrm{IK}}}\ (\mathsf{box}\ t)\ e \sim \mathit{wk}\ (\mathit{factor}\ e)\ t} \qquad \Box\text{-}\eta\quad \frac{\Gamma \vdash t : \Box A}{\Gamma \vdash t \sim \mathsf{box}\ (\mathsf{unbox}_{\lambda_{\mathrm{IK}}}\ t\ \mathsf{nil})}$$

Fig. 7. Equational theory for λ_{IK}

3.1.2 Possible-World Semantics. A possible-world model is defined using the notion of a possible-world frame as below. We work in a constructive type-theoretic metalanguage, and denote the universe of types in this language by *Type*.

Definition 1 (Possible-world frame). A frame F is given by a triple (W, R_i, R_m) consisting of a type W : *Type* and two relations R_i and R_m: $W \times W \to$ *Type* on W such that the following conditions are satisfied:

- R_i is *reflexive* and *transitive*
- if $w\ R_m\ v$ and $v\ R_i\ v'$ then there exists some w' : W such that $w\ R_i\ w'$ and $w'\ R_m\ v'$; this *factorization* condition can be pictured as an implication $R_m\ ;\ R_i \subseteq R_i\ ;\ R_m$ or diagrammatically as follows:



(note that neither w' nor the proofs of relatedness are required to be unique, nor will they all be in the frames that we will consider)

Definition 2 (Possible-world model). A possible-world model $\mathcal{M}$ is given by a tuple (F, V) consisting of a frame F (see Definition 1) and a W-indexed family $V_\iota : W \to$ *Type* (called the *valuation* of the base type) such that $\forall w, w'.\ w\ R_i\ w' \to V_{\iota,w} \to V_{\iota,w'}$.

We have omitted coherence conditions from these definitions for readability. Those conditions stem from the proof relevance of the relations and predicates involved. They will be satisfied by the models we will construct, and will also be given below for completeness.

The types and typing contexts in λ_{IK} are interpreted in a possible-world model via the interpretation functions $[\![-]\!]$ defined in Section 2. To evaluate terms, we must first prove the following *monotonicity* lemma. This lemma is well-known as a requirement

to give a sound interpretation of the function type in an arbitrary possible-world model, and can be thought of as the semantic generalization of renaming in terms.

Lemma 1 (Monotonicity). *In every possible-world model $\mathcal{M}$, for every type A and worlds w and w', we have a function $wk_A : w\ R_i\ w' \to [\![A]\!]_w \to [\![A]\!]_{w'}$. And similarly, for every context Γ, a function $wk_\Gamma : w\ R_i\ w' \to [\![\Gamma]\!]_w \to [\![\Gamma]\!]_{w'}$.*

We evaluate terms in λ_{IK} in a possible-world model as follows.

$$
\begin{aligned}
&[\![-]\!] : \Gamma \vdash A \to (\forall w.\, [\![\Gamma]\!]_w \to [\![A]\!]_w)\\
&[\![\mathsf{var}\, v \qquad\quad]\!]\ \gamma = \mathit{lookup}\ v\ \gamma\\
&[\![\lambda\, t \qquad\quad]\!]\ \gamma = \lambda i.\, \lambda a.\, [\![t]\!]\ (\mathit{wk}\ i\ \gamma, a)\\
&[\![\mathsf{app}\, t\, u \quad]\!]\ \gamma = ([\![t]\!]\ \gamma)\ \mathsf{id}_{\leq}\ ([\![u]\!]\ \gamma)\\
&[\![\mathsf{box}\, t \qquad\quad]\!]\ \gamma = \lambda i.\, \lambda m.\, [\![t]\!]\ (\mathit{wk}\ i\ \gamma, m)\\
&[\![\mathsf{unbox}_{\lambda_{\text{IK}}}\, t\, e]\!]\ \gamma = [\![t]\!]\ \delta\ \mathsf{id}_{\leq}\ m\\
&\qquad \text{where } (\delta, m) = \mathit{trim}_{\lambda_{\text{IK}}}\ \gamma\ e
\end{aligned}
$$

The evaluation of terms in the simply-typed fragment is standard, and resembles the evaluator of STLC. Variables are interpreted by a lookup function that projects values from an environment, and λ-abstraction and application are evaluated using their semantic counterparts. To evaluate λ-abstraction, we must construct a semantic function $\forall w'.\, w\ R_i\ w' \to [\![A]\!]_{w'} \to [\![B]\!]_{w'}$ using the given term $\Gamma, A \vdash t : B$ and environment $\gamma : [\![\Gamma]\!]_w$. We achieve this by recursively evaluating t in an environment that extends γ appropriately using the semantic arguments $i : w\ R_i\ w'$ and $a : [\![A]\!]_{w'}$. We use the monotonicity lemma to "transport" $[\![\Gamma]\!]_w$ to $[\![\Gamma]\!]_{w'}$, and construct an environment of type $[\![\Gamma]\!]_{w'} \times [\![A]\!]_{w'}$ for recursively evaluating t, which produces the desired result of type $[\![B]\!]_{w'}$. Application is evaluated by simply recursively evaluating the applied terms and applying them in the semantics with a value $\mathsf{id}_{\leq} : w\ R_i\ w$, which is available since R_i is reflexive.

In the modal fragment, to evaluate the term $\Gamma \vdash \mathsf{box}\, t : \Box A$ with $\gamma : [\![\Gamma]\!]_w$, we must construct a function of type $\forall w'.\, w\ R_i\ w' \to \forall v.\, w'\ R_m\ v \to [\![A]\!]_v$. Using the semantic arguments $i : w\ R_i\ w'$ and $m : w'\ R_m\ v$, we recursively evaluate the term $\Gamma, \locked \vdash t : A$ in the extended environment $(\mathit{wk}\ i\ \gamma, m) : [\![\Gamma, \locked]\!]_v$, since $[\![\Gamma, \locked]\!]_v = \sum_{w'} [\![\Gamma]\!]_{w'} \times w'\ R_m\ v$. On the other hand, the term $\Gamma \vdash \mathsf{unbox}_{\lambda_{\text{IK}}}\, t\, e : A$ with $e : \Delta \lhd_{\lambda_{\text{IK}}} \Gamma$ and $\Delta \vdash t : \Box A$, for some Δ, must be evaluated with an environment $\gamma : [\![\Gamma]\!]_w$. To recursively evaluate the term $\Delta \vdash t : \Box A$, we must first discard the part of the environment γ that substitutes the types in the extension of $\Delta, \locked$. This is achieved using the function $\mathit{trim}_{\lambda_{\text{IK}}} : [\![\Gamma]\!]_w \to \Delta \lhd_{\lambda_{\text{IK}}} \Gamma \to [\![\Delta, \locked]\!]_w$ that projects γ to produce an environment $\delta : [\![\Delta]\!]_{v'}$ and a value $m : v'\ R_m\ w$. We evaluate t with δ and apply the resulting function of type $\forall v.\, v\ R_i\ v' \to \forall w.\, v'\ R_m\ w \to [\![A]\!]_w$ to $\mathsf{id}_{\leq}$ and m to return the desired result.

We state the soundness of λ_{IK} with respect to the possible-world semantics before we instantiate it with the NbE model that we will construct in the next subsection. We note that the soundness proof relies on the possible-world models to satisfy coherence conditions that we have omitted from Definitions 1 and 2 but that will be satisfied by the NbE models. Specifically, W and R_i together with the transitivity and reflexivity proofs trans_i and refl_i for R_i need to form a category $\mathscr{W}$, i.e. trans_i needs to be associative and refl_i needs to be a unit for trans_i; the proofs of the factorization condition

need to satisfy the functoriality laws $factor_i\ m\ (refl_i\ v) = refl_i\ w$, $factor_m\ m\ (refl_i\ v) = m$, $factor_i\ m\ (trans_i\ i\ j) = trans_i\ (factor_i\ m\ i)\ (factor_i\ m'\ j)$ and $factor_m\ m\ (trans_i\ i\ j) = factor_m\ m'\ j$ where $m' \coloneqq factor_m\ m\ i : w'\ R_m\ v'$ denotes the modal accessibility proof produced by the first factorization of $m : w\ R_m\ v$ and $i : v\ R_i\ v'$; and V_ι together with the monotonicity proof wk_ι needs to form a functor on the category $\mathscr{W}$, i.e. $wk_\iota\ (refl_i\ w)$ needs to be equal to the identity function on $V_{\iota,w}$ and $wk_\iota\ (trans_i\ i\ j)$ needs to be equal to the composite $wk_\iota\ j \circ wk_\iota\ i$.

Theorem 2. *Let $\mathcal{M}$ be any possible-world model (see Definition 2). If two terms t and $u : \Gamma \vdash A$ of λ_{IK} are equivalent (see Fig. 7) then the functions $[\![t]\!]$ and $[\![u]\!] : \forall w.\ [\![\Gamma]\!]_w \to [\![A]\!]_w$ as determined by $\mathcal{M}$ are equal.*

Proof. Let $\mathcal{M}$ be a possible-world model with underlying frame $F = (W, R_i, R_m)$. Denote the category whose objects are worlds $w : W$ and whose morphisms are proofs $i : w\ R_i\ w'$ by C. The frame F can be seen as determining an adjunction $\blacksquare \dashv \Box$ on the category of presheaves indexed by the category C, which is moreover well-known to be Cartesian closed. The interpretation $[\![-]\!]$ can then be seen as factoring through the categorical semantics described in Clouston [12, Section 2.3], of which the category of presheaves over C is an instance by virtue of its Cartesian closure and equipment with an adjunction. We can therefore conclude by applying Clouston [12, Theorem 2.8 (Categorical Soundness) and remark below that]. □

3.1.3 NbE Model. The normal forms of terms in λ_{IK} are defined along with neutral elements in a mutually recursive fashion by the judgements $\Gamma \vdash_{NF} A$ and $\Gamma \vdash_{NE} A$, respectively, in Fig. 8. Intuitively, a normal form may be thought of as a value, and a neutral element may be thought of as a "stuck" computation. We extend the standard definition of normal forms and neutral elements in STLC with Rules Nf/□-Intro and λ_{IK}/Ne/□-Elim.

$$\textsc{Ne/Var}\ \frac{\Gamma \vdash_{VAR} v : A}{\Gamma \vdash_{NE} \mathsf{var}\, v : A} \qquad \textsc{Nf/Up}\ \frac{\Gamma \vdash_{NE} n : \iota}{\Gamma \vdash_{NF} \mathsf{up}\, n : \iota} \qquad \textsc{Nf/}\Rightarrow\textsc{-Intro}\ \frac{\Gamma, A \vdash_{NF} n : B}{\Gamma \vdash_{NF} \lambda\, n : A \Rightarrow B}$$

$$\textsc{Ne/}\Rightarrow\textsc{-Elim}\ \frac{\Gamma \vdash_{NE} n : A \Rightarrow B \qquad \Gamma \vdash_{NF} m : A}{\Gamma \vdash_{NE} \mathsf{app}\, n\, m : B} \qquad \textsc{Nf/}\Box\textsc{-Intro}\ \frac{\Gamma, \blacksquare \vdash_{NF} n : A}{\Gamma \vdash_{NF} \mathsf{box}\, n : \Box A}$$

$$\lambda_{IK}\textsc{/Ne/}\Box\textsc{-Elim}\ \frac{\Delta \vdash_{NE} n : \Box A \qquad e : \Delta \lhd_{\lambda_{IK}} \Gamma}{\Gamma \vdash_{NE} \mathsf{unbox}_{\lambda_{IK}}\, n\, e : A}$$

Fig. 8. Normal forms and neutral elements in λ_{IK}

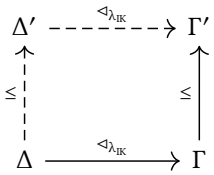
Recall that an NbE model for a given calculus C is a particular kind of model $\mathcal{M}$ that comes equipped with a function $quote : \mathcal{M}([\![\Gamma]\!], [\![A]\!]) \to \Gamma \vdash_{NF} A$ satisfying $t \sim quote\, [\![t]\!]$ for all terms $t : \Gamma \vdash A$ where $[\![-]\!]$ denotes the *generic* evaluation function for C.

Using the relations defined in Figs. 3 and 4, we construct an NbE model for λ_{IK} by instantiating the parameters that define a *possible-world* model as follows.

- Worlds as contexts: $W = \mathit{Ctx}$
- Relation R_i as order-preserving embeddings: $\Gamma\ R_i\ \Gamma' = \Gamma \leq \Gamma'$
- Relation R_m as extensions of a "locked" context: $\Delta\ R_m\ \Gamma = \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$
- Valuation V_ι as neutral elements: $V_{\iota,\Gamma} = \Gamma \vdash_{\mathrm{NE}} \iota$

The condition that the valuation must satisfy $\mathit{wk}_A : \Gamma \leq \Gamma' \to \Gamma \vdash_{\mathrm{NE}} A \to \Gamma' \vdash_{\mathrm{NE}} A$, for all types A, can be shown by induction on the neutral term $\Gamma \vdash_{\mathrm{NE}} A$. To show that this model is indeed a possible-world model, it remains for us to show that the frame conditions are satisfied.

The first frame condition states that OPEs must be reflexive and transitive, which can be shown by structural induction on the context and definition of OPEs, respectively. The second frame condition states that given $\Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$ and $\Gamma \leq \Gamma'$ there is a $\Delta' : \mathit{Ctx}$ such that $\Delta \leq \Delta'$ and $\Delta' \lhd_{\lambda_{\mathrm{IK}}} \Gamma'$,



which can be shown by constructing a function by simultaneous recursion on OPEs and the modal accessibility relation.

Observe that the instantiation of the monotonicity lemma in the NbE model states that we have the functions $\mathit{wk}_A : \Gamma \leq \Gamma' \to [\![A]\!]_\Gamma \to [\![A]\!]_{\Gamma'}$ and $\mathit{wk}_\Delta : \Gamma \leq \Gamma' \to [\![\Delta]\!]_\Gamma \to [\![\Delta]\!]_{\Gamma'}$, which allow denotations of types and contexts to be renamed with respect to an OPE.

To implement the function *quote*, we first implement *reification* and *reflection*, using two functions $\mathit{reify}_A : [\![A]\!]_\Gamma \to \Gamma \vdash_{\mathrm{NF}} A$ and $\mathit{reflect}_A : \Gamma \vdash_{\mathrm{NE}} A \to [\![A]\!]_\Gamma$, respectively. Reification converts a semantic value to a normal form, while reflection converts a neutral element to a semantic value. They are implemented as follows by induction on the index type A.

$$
\begin{aligned}
&\mathit{reify}_{A,\Gamma} : [\![A]\!]_\Gamma \to \Gamma \vdash_{\mathrm{NF}} A \\
&\mathit{reify}_{\iota,\Gamma}\quad n = \mathsf{up}\, n \\
&\mathit{reify}_{A\Rightarrow B,\Gamma}\ f = \lambda\,(\mathit{reify}_{B,(\Gamma,A)}\,(f\ (\mathsf{drop}\ \mathsf{id}_\leq)\ \mathit{fresh}_{A,\Gamma})) \\
&\mathit{reify}_{\Box A,\Gamma}\quad b = \mathsf{box}\,(\mathit{reify}_{A,(\Gamma,🔒)}\,(b\ \mathsf{id}_\leq\ \mathsf{nil}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reflect}_{A,\Gamma} : \Gamma \vdash_{\mathrm{NE}} A \to [\![A]\!]_\Gamma \\
&\mathit{reflect}_{\iota,\Gamma}\quad n = n \\
&\mathit{reflect}_{A\Rightarrow B,\Gamma}\ n = \lambda(o : \Gamma \leq \Gamma').\,\lambda a.\,\mathit{reflect}_{B,\Gamma}(\mathsf{app}\ (\mathit{wk}_{A\Rightarrow B}\ o\ n)\ (\mathit{reify}_{A,\Gamma'}\ a)) \\
&\mathit{reflect}_{\Box A,\Gamma}\quad n = \lambda(o : \Gamma \leq \Gamma').\,\lambda(e : \Gamma' \lhd_{\lambda_{\mathrm{IK}}} \Delta).\,\mathit{reflect}_{A,\Delta}(\mathsf{unbox}_{\lambda_{\mathrm{IK}}}\ (\mathit{wk}_{\Box A}\ o\ n)\ e)
\end{aligned}
$$

For the function type, we recursively reify the body of the λ-abstraction by applying the given semantic function f with suitable arguments, which are an OPE $\mathsf{drop}\ \mathsf{id}_\leq : \Gamma \leq \Gamma, A$ and a value $\mathit{fresh}_{A,\Gamma} = \mathit{reflect}_{A,(\Gamma,A)}\ (\mathsf{var}\ \mathsf{zero}) : [\![A]\!]_{\Gamma,A}$—which is the De Bruijn index equivalent of a fresh variable. Reflection, on the other hand, recursively reflects the application of a neutral $\Gamma \vdash_{\mathrm{NE}} n : A \Rightarrow B$ to the reification of the semantic

argument $a : [\![A]\!]_{\Gamma'}$ for an OPE $o : \Gamma \leq \Gamma'$. Similarly, for the $\Box$ type, we recursively reify the body of box by applying the given semantic function $b : \forall \Gamma.\, \Gamma \leq \Gamma' \to \forall \Delta.\, \Gamma' \lhd_{\lambda_{\text{IK}}} \Delta \to [\![A]\!]_\Delta$ to suitable arguments $\text{id}_\leq : \Gamma \leq \Gamma$ and the empty context extension $\text{nil} : \Gamma \lhd_{\lambda_{\text{IK}}} \Gamma, \text{🔒}$. Reflection also follows a similar pursuit by reflecting the application of the neutral $\Gamma \vdash_{\text{NE}} n : \Box A$ to the eliminator unbox.

Equipped with reification, we implement *quote* (as seen below), by applying the given denotation of a term, a function $f : \forall \Delta.\, [\![\Gamma]\!]_\Delta \to [\![A]\!]_\Delta$, to the identity environment $\mathit{freshEnv}_\Gamma : [\![\Gamma]\!]_\Gamma$, and then reifying the resulting value. The construction of the value $\mathit{freshEnv}_\Gamma$ is the De Bruijn index equivalent of generating an environment with fresh variables.

$$
\begin{aligned}
&\mathit{quote} : (\forall \Delta.\, [\![\Gamma]\!]_\Delta \to [\![A]\!]_\Delta) \to \Gamma \vdash_{\text{NF}} A \\
&\mathit{quote}\ f = \mathit{reify}_{A,\Gamma}\ (f\ \mathit{freshEnv}_\Gamma)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{freshEnv}_\Gamma : [\![\Gamma]\!]_\Gamma \\
&\mathit{freshEnv}_{\cdot} = () \\
&\mathit{freshEnv}_{\Gamma,A} = (\mathit{wk}\ (\text{drop}\ \text{id}_\leq)\ \mathit{freshEnv}_\Gamma, \mathit{fresh}_{A,\Gamma}) \\
&\mathit{freshEnv}_{\Gamma,\text{🔒}} = (\mathit{freshEnv}_\Gamma, \text{nil})
\end{aligned}
$$

To prove that the function *quote* is indeed a retraction of evaluation, we follow the usual logical relations approach. As seen in Fig. 9, we define a relation L_A indexed by a type A that relates a term $\Gamma \vdash t : A$ to its denotation $a : [\![A]\!]_\Gamma$ as $L_A\ t\ a$. From a proof of $L_A\ t\ a$, it can be shown that $t \sim \mathit{reify}_A\ a$. This relation is extended to contexts as L_Δ, for some context Δ, which relates a substitution $\Gamma \vdash s : \Delta$ to its denotation $\delta : [\![\Delta]\!]_\Gamma$ as $L_\Delta\ s\ \delta$.

$$
\begin{aligned}
&L_{A,\Gamma} : \Gamma \vdash A \to [\![A]\!]_\Gamma \to \mathit{Type} \\
&L_{\iota,\Gamma}\quad t\ n = t \sim \mathit{quote}\ n \\
&L_{A \Rightarrow B,\Gamma}\ t\ f = \forall \Gamma', o : \Gamma \leq \Gamma', u, a.\ L_{A,\Gamma'}\ u\ a \to L_{B,\Gamma'}\ (\text{app}\ (\mathit{wk}\ o\ t)\ u)\ (f\ o\ a) \\
&L_{\Box A,\Gamma}\quad t\ b = \forall \Gamma', o : \Gamma \leq \Gamma', e : \Gamma' \lhd_{\lambda_{\text{IK}}} \Delta.\ L_{A,\Delta}\ (\text{unbox}_{\lambda_{\text{IK}}}\ (\mathit{wk}\ o\ t)\ e)\ (b\ o\ e)
\end{aligned}
$$

$$
\begin{aligned}
&L_{\Delta,\Gamma} : \Gamma \vdash_{\text{s}} \Delta \to [\![\Delta]\!]_\Gamma \to \mathit{Type} \\
&L_{\cdot,\Gamma}\quad \text{empty}\qquad\qquad\qquad\quad () = \top \\
&L_{(\Delta,A),\Gamma}\ (\text{ext}\ s\ t)\qquad\quad (\delta, a) = L_{\Delta,\Gamma}\ s\ \delta \times L_{A,\Gamma}\ t\ a \\
&L_{(\Delta,\text{🔒}),\Gamma}\ (\text{ext}_\text{🔒}\ s\ (e : \Theta \lhd_{\lambda_{\text{IK}}} \Gamma))\ (\delta, e) = L_{\Delta,\Theta}\ s\ \delta
\end{aligned}
$$

Fig. 9. Logical relations for λ_{IK}

For the logical relations, we then prove the so-called fundamental theorem.

Proposition 3 (Fundamental theorem). *Given a term $\Delta \vdash t : A$, a substitution $\Gamma \vdash_s s : \Delta$ and a value $\delta : [\![\Delta]\!]_\Gamma$, if $L_{\Delta,\Gamma}\ s\ \delta$ then $L_{A,\Gamma}\ (\mathit{subst}\ s\ t)\ ([\![t]\!]\ \delta)$.*

We conclude this subsection by stating the normalization theorem for λ_{IK}.

Proposition 3 entails that $L_{A,\Delta}\ (\mathit{subst}\ \text{id}_\text{s}\ t)\ ([\![t]\!]\ \mathit{freshEnv}_\Delta)$ for any term t, if we pick s as the identity substitution $\text{id}_\text{s} : \Delta \vdash_\text{s} \Delta$, and δ as $\mathit{freshEnv}_\Delta : [\![\Delta]\!]_\Delta$, since they can be shown to be related as $L_{\Delta,\Delta}\ \text{id}_\text{s}\ \mathit{freshEnv}_\Delta$. From this it follows that $\mathit{subst}\ \text{id}_\text{s}\ t \sim \mathit{reify}_A\ ([\![t]\!]\ \mathit{freshEnv}_\Delta)$, and further that $t \sim \mathit{quote}\ [\![t]\!]$ from the definition of *quote*

and the fact that $\mathit{subst}\,\mathsf{id}_{\mathsf{s}}\,t = t$. As a result, the composite $\mathit{norm} = \mathit{quote} \circ [\![-]\!]$ is *adequate*, i.e. $\mathit{norm}\,t = \mathit{norm}\,t'$ implies $t \sim t'$.

The soundness of λ_{IK} with respect to possible-world models (see Theorem 2) directly entails $\mathit{quote}\,[\![t]\!] = \mathit{quote}\,[\![u]\!] : \Gamma \vdash_{\mathrm{NF}} A$ for all terms $t, u : \Gamma \vdash A$ such that $\Gamma \vdash t \sim u : A$, which means that $\mathit{norm} = \mathit{quote} \circ [\![-]\!]$ is *complete*. Note that this terminology might be slightly confusing because it is the *soundness* of $[\![-]\!]$ that implies the *completeness* of *norm*.

Theorem 4. *Let $\mathcal{M}$ denote the possible-world model over the frame given by the relations $\Gamma \leq \Gamma'$ and $\Delta \lhd_{\lambda_{IK}} \Gamma$ and the valuation $V_{\iota,\Gamma} = \Gamma \vdash_{NE} \iota$.*

There is a function $\mathit{quote} : \mathcal{M}([\![\Gamma]\!], [\![A]\!]) \to \Gamma \vdash_{NF} A$ such that the composite $\mathit{norm} = \mathit{quote} \circ [\![-]\!] : \Gamma \vdash A \to \Gamma \vdash_{NF} A$ from terms to normal forms of λ_{IK} is complete and adequate.

3.2 Extending to the Calculus $\lambda_{\mathbf{IS4}}$

3.2.1 Terms, Substitutions and Equational Theory. To define the intrinsically-typed syntax of λ_{IS4}, we first define the modal accessibility relation on contexts in Fig. 10.

$$\mathsf{nil} : \Gamma \lhd_{\lambda_{\mathrm{IS4}}} \Gamma \qquad \frac{e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma}{\mathsf{var}\,e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma, A} \qquad \frac{e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma}{\mathsf{lock}\,e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma, \blacksquare}$$

Fig. 10. Modal accessibility relation on contexts (λ_{IS4})

If $\Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma$ then Γ is an extension of Δ with as many locks as needed. Note that, in contrast to λ_{IK}, the modal accessibility relation is both reflexive and transitive. This corresponds to the conditions on the accessibility relation for the logic IS4.

Fig. 11 presents the changes of λ_{IK} that yield λ_{IS4}. The terms are the same as λ_{IK} with the exception of Rule $\lambda_{\mathrm{IK}}/\Box$-Elim which now includes the modal accessibility relation for λ_{IS4}. Similarly, the substitution rule for contexts with locks now refers to $\lhd_{\lambda_{\mathrm{IS4}}}$.

$$\frac{\Delta \vdash t : \Box A \qquad e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma}{\Gamma \vdash \mathsf{unbox}_{\lambda_{\mathrm{IS4}}}\,t\,e : A}\ \lambda_{\mathrm{IS4}}/\Box\text{-Elim} \qquad \frac{\Theta \vdash s : \Delta \qquad e : \Theta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma}{\Gamma \vdash_{\mathsf{s}} \mathsf{ext}_{\blacksquare}\,s\,e : \Delta, \blacksquare}$$

Fig. 11. Intrinsically-typed terms and substitutions of λ_{IS4} (omitting the unchanged rules of Fig. 5)

Fig. 12 presents the equational theory of the modal fragment of λ_{IS4}. This is a slightly modified version of λ_{IK} (cf. Fig. 7) that accommodates the changes to the rule $\lambda_{\mathrm{IS4}}/\Box$-Elim. Unlike before, Rule $\Box$-β now performs a substitution to modify the term $\Delta, \blacksquare \vdash t : A$ to a term of type $\Gamma \vdash A$. Note that the result of such a substitution need not yield the same term since substitution may change the context extension of some subterm.

$$\Box\text{-}\beta\quad \frac{\Delta, \blacksquare \vdash t : A \qquad e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma}{\Gamma \vdash \mathsf{unbox}_{\lambda_{\mathrm{IS4}}}\ (\mathsf{box}\ t)\ e \sim \mathit{subst}\ (\mathsf{ext}_{\blacksquare}\ \mathsf{id}_{\mathsf{s}}\ e)\ t} \qquad \Box\text{-}\eta\quad \frac{\Gamma \vdash t : \Box A}{\Gamma \vdash t \sim \mathsf{box}\ (\mathsf{unbox}_{\lambda_{\mathrm{IS4}}}\ t\ (\mathsf{lock}\ \mathsf{nil}))}$$

Fig. 12. Equational theory for λ_{IS4} (omitting the unchanged rules of Fig. 7)

3.2.2 Possible-World Semantics. Giving possible-world semantics for λ_{IS4} requires an additional frame condition on the relation R_m: it must be reflexive and transitive. Evaluation proceeds as before, where we use a function $\mathit{trim}_{\lambda_{\mathrm{IS4}}} : \forall w.\ [\![\Gamma]\!]_w \to \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma \to [\![\Delta, \blacksquare]\!]_w$ to manipulate the environment for evaluating $\mathsf{unbox}_{\lambda_{\mathrm{IS4}}}\ t\ e$, as seen below.

$$[\![\mathsf{unbox}_{\lambda_{\mathrm{IS4}}}\ t\ e]\!]\ \gamma = [\![t]\!]\ \delta\ \mathsf{id}_{\leq}\ m \quad \text{where } (\delta, m) = \mathit{trim}_{\lambda_{\mathrm{IS4}}}\ \gamma\ e$$

The additional frame requirements ensures that the function $\mathit{trim}_{\lambda_{\mathrm{IS4}}}$ can be implemented. For example, consider implementing the case of $\mathit{trim}_{\lambda_{\mathrm{IS4}}}$ for some argument of type $[\![\Gamma]\!]_w$ and the extension $\mathsf{nil} : \Gamma \lhd_{\lambda_{\mathrm{IS4}}} \Gamma$ that adds zero locks. The desired result is of type $[\![\Gamma, \blacksquare]\!]_w$, which is defined as $\sum_v [\![\Gamma]\!]_v \times v\ R_m\ w$. We construct such a result using the argument of $[\![\Gamma]\!]_w$ by picking v as w itself, and using the reflexivity of R_m to construct a value of type $w\ R_m\ w$. Similarly, the transitivity of R_m is required when the context extension adds more than one lock.

Analogously to Theorem 2, we state the soundness of λ_{IS4} with respect to *reflexive and transitive* possible-world models before we instantiate it with the NbE model that we will construct in the next subsection. In addition to the coherence conditions stated before Theorem 2 the soundness proof for λ_{IS4} relies on coherence conditions involving the additional proofs refl_m and trans_m that a reflexive and transitive modal accessibility relation R_m must come equipped with. Specifically, trans_m also needs to be associative, refl_m also needs to be a unit for trans_m, and the proofs of the factorization condition also need to satisfy the functoriality laws in the modal accessibility argument, i.e. $\mathit{factor}_i\ (\mathit{refl}_m\ w)\ i = i$, $\mathit{factor}_m\ (\mathit{refl}_m\ w)\ i = \mathit{refl}_m\ w'$, $\mathit{factor}_i\ (\mathit{trans}_m\ n\ m)\ i = \mathit{factor}_i\ n\ i'$ and $\mathit{factor}_m\ (\mathit{trans}_m\ n\ m)\ i = \mathit{trans}_m\ (\mathit{factor}_m\ n\ i')\ (\mathit{factor}_m\ m\ i)$ where $i' := \mathit{factor}_i\ m\ i : w\ R_i\ w'$.

Proposition 5. *Let $\mathcal{C}$ be a Cartesian closed category equipped with a comonad $\Box$ that has a left adjoint $\blacksquare \dashv \Box$, then equivalent terms t and $u : \Gamma \vdash A$ denote equal morphisms in $\mathcal{C}$.*

Proof. This is a version of Clouston [12, Theorem 4.8] for λ_{IS4} where the side condition of Rule $\lambda_{\mathrm{IS4}}/\Box$-Elim appears as an argument to the term former unbox and hence idempotency is not imposed on the comonad $\Box$. □

Theorem 6. *Let $\mathcal{M}$ be a possible-world model (see Definition 2) such that the modal accessibility relation R_m is reflexive and transitive. If two terms t and $u : \Gamma \vdash A$ of λ_{IS4} are equivalent (see Fig. 12) then the functions $[\![t]\!]$ and $[\![u]\!] : \forall w.\ [\![\Gamma]\!]_w \to [\![A]\!]_w$ as determined by $\mathcal{M}$ are equal.*

Proof. The right adjoint determined by a reflexive and transitive frame has a comonad structure so that we can conclude by applying Proposition 5. □

3.2.3 NbE Model. The normal forms of λ_{IS4} are defined as before, except for the following rule replacing the neutral rule λ_{IK}/Ne/$\Box$-Elim.

$$\lambda_{\text{IS4}}\text{/Ne/}\Box\text{-Elim}\quad \frac{\Delta \vdash_{\text{NE}} n : \Box A \qquad e : \Delta \lhd_{\lambda_{\text{IS4}}} \Gamma}{\Gamma \vdash_{\text{NE}} \mathsf{unbox}_{\lambda_{\text{IS4}}}\, n\, e : A}$$

The NbE model construction also proceeds in the same way, where we now pick the relation R_m as arbitrary extensions of a context: $\Delta\ R_m\ \Gamma = \Delta \lhd_{\lambda_{\text{IS4}}} \Gamma$. The modal fragment for *reify* and *reflect* are now implemented as follows:

$$\begin{aligned} &\mathit{reify}_{\Box A,\Gamma}\quad b = \mathsf{box}\,(\mathit{reify}_{A,(\Gamma,🔒)}\,(b\,\mathsf{id}_{\leq}\,(\mathsf{lock}\,\mathsf{nil}))) \\ &\mathit{reflect}_{\Box A,\Gamma}\ n = \lambda(o : \Gamma \leq \Gamma').\,\lambda(e : \Gamma' \lhd_{\lambda_{\text{IS4}}} \Delta).\,\mathit{reflect}_{A,\Delta}\,(\mathsf{unbox}\,(\mathit{wk}\,o\,n)\,e) \end{aligned}$$

Theorem 7. *Let $\mathcal{M}$ denote the possible-world model over the reflexive and transitive frame given by the relations $\Gamma \leq \Gamma'$ and $\Delta \lhd_{\lambda_{IS4}} \Gamma$ and the valuation $V_{\iota,\Gamma} = \Gamma \vdash_{NE} \iota$.*

There is a function quote $: \mathcal{M}(\llbracket\Gamma\rrbracket, \llbracket A\rrbracket) \to \Gamma \vdash_{NF} A$ such that the composite norm $=$ quote $\circ\ \llbracket - \rrbracket : \Gamma \vdash A \to \Gamma \vdash_{NF} A$ from terms to normal forms of λ_{IS4} is complete and adequate.

The proof of this theorem requires us to identify terms by extending the equational theory of λ_{IS4} with an additional rule. To understand the need for it, consider unboxing a term $\Gamma \vdash t : \Box A$ into an extended context Γ, B in λ_{IS4}. We may first weaken t as $\Gamma, B \vdash \mathit{wk}\,(\mathsf{drop}\,\mathsf{id}_{\leq})\,t : \Box A$ and then apply unbox as $\Gamma, B \vdash \mathsf{unbox}\,(\mathit{wk}\,(\mathsf{drop}\,\mathsf{id}_{\leq})\,t)\,\mathsf{nil} : A$. However, we may also apply unbox on t as $\Gamma, B \vdash \mathsf{unbox}\,t\,(\mathsf{var}\,\mathsf{nil}) : A$. This weakens the term "explicitly" in the sense that the weakening with B is recorded in the term by the proof var nil of the modal accessibility relation $\Gamma \lhd_{\lambda_{\text{IS4}}} \Gamma, B$. The two ways of unboxing $\Gamma \vdash t : \Box A$ into the extended context Γ, B result in two terms with the same denotation in the possible-world semantics but *distinct* typing derivations. We wish the two typing derivations $\mathsf{unbox}\,t\,(\mathsf{var}\,\mathsf{nil})$ and $\mathsf{unbox}\,(\mathit{wk}\,(\mathsf{drop}\,\mathsf{id}_{\leq})\,t)\,\mathsf{nil}$ to be identified. For this reason, we extend the equational theory of λ_{IS4} with the rule $\mathsf{unbox}\,t\,(\mathit{trans}_m\,e\,e') \sim \mathsf{unbox}\,(\mathit{wk}\,(\mathit{toOPE}\,e)\,t)\,e'$ for any *lock-free* extension e, which can be converted to a sequence of drops using the function *toOPE*. Explicit weakening can also be avoided by, instead of extending the equational theory, changing the definition of the modal accessibility relation such that $\Delta \lhd_{\lambda_{\text{IS4}}} \Gamma$ holds only if $\Gamma = \Delta$ or $\Gamma = \Delta, 🔒, \Gamma'$ for some Γ'. Note that the modal accessibility relation for λ_{IK}, where the issue of explicit weakening does not occur, satisfies this property.

3.3 Extending to the Calculi λ_{IT} and λ_{IK4}

The NbE model construction for λ_{IT} and λ_{IK4} follows a similar pursuit as λ_{IS4}. We define suitable modal accessibility relations $\lhd_{\lambda_{\text{IT}}}$ and $\lhd_{\lambda_{\text{IK4}}}$ as extensions that allow the addition of at most one 🔒, and at least one lock 🔒, respectively. To give possible-world semantics, we require an additional frame condition that the relation R_m be reflexive for λ_{IT} and transitive for λ_{IK4}. For evaluation, we use a function $\mathit{trim}_{\lambda_{\text{IT}}} : \llbracket\Gamma\rrbracket_w \to \Delta \lhd_{\lambda_{\text{IT}}} \Gamma \to \llbracket\Delta, 🔒\rrbracket_w$ for λ_{IT}, and similarly $\mathit{trim}_{\lambda_{\text{IK4}}}$ for λ_{IK4}. The modification to the neutral rule λ_{IK}/Ne/$\Box$-Elim is achieved as before in λ_{IS4} using the corresponding modal accessibility relations. Unsurprisingly, reification and reflection can also be implemented, thus yielding normalization functions for both λ_{IT} and λ_{IK4}.

4 COMPLETENESS, DECIDABILITY AND LOGICAL APPLICATIONS

In this section we record some immediate consequences of the model constructions we presented in the previous section.

Completeness of the Equational Theory. As a corollary of the adequacy of an NbE model $\mathcal{N}$, i.e. $\Gamma \vdash t \sim u : A$ whenever $[\![t]\!] = [\![u]\!] : \mathcal{N}([\![\Gamma]\!], [\![A]\!])$, we obtain completeness of the equational theory with respect to the class of models that the respective NbE model belongs to. Given the NbE models constructed in Subsections 3.1.3 and 3.2.3 this means that the equational theories of λ_{IK} and λ_{IS4} (cf. Fig. 7) are (sound and) complete with respect to the class of Cartesian closed categories equipped with an adjunction and a right-adjoint comonad, respectively.

Theorem 8. *Let $t, u : \Gamma \vdash A$ be two terms of λ_{IK}. If for all Cartesian closed categories $\mathcal{M}$ equipped with an adjunction it is the case that $[\![t]\!] = [\![u]\!] : \mathcal{M}([\![\Gamma]\!], [\![A]\!])$ then $\Gamma \vdash t \sim u : A$.*

Proof. Let $\mathcal{M}_0$ be the model we constructed in Subsection 3.1.3. Since $\mathcal{M}_0$ is a Cartesian closed category equipped with an adjunction, by assumption we have $[\![t]\!]_{\mathcal{M}_0} = [\![u]\!]_{\mathcal{M}_0}$. And lastly, since $\mathcal{M}_0$ is an NbE model, we have $\Gamma \vdash t \sim \mathit{quote}\, [\![t]\!]_{\mathcal{M}_0} = \mathit{quote}\, [\![u]\!]_{\mathcal{M}_0} \sim u : A$. □

Note that this statement corresponds to Clouston [12, Theorem 3.2] but there it is obtained via a term model construction and for the term model to be equipped with an adjunction the calculus needs to be first extended with an internalization of the operation 🔒 on contexts as an operation ♦ on types.

Theorem 9. *Let $t, u : \Gamma \vdash A$ be two terms of λ_{IS4}. If for all Cartesian closed categories $\mathcal{M}$ equipped with a right-adjoint comonad it is the case that $[\![t]\!] = [\![u]\!] : \mathcal{M}([\![\Gamma]\!], [\![A]\!])$ then $\Gamma \vdash t \sim u : A$.*

Proof. As for Theorem 8. □

This statement corresponds to Clouston [12, Section 4.4] but there it is proved for an equational theory that identifies terms up to differences in the accessibility proofs and with respect to the class of models where the comonad is *idempotent*, to which the model of Subsection 3.2.3 does not belong.

Completeness of the Deductive Theory. Using the quotation function of an NbE model $\mathcal{N}$, i.e. $\mathit{quote} : \mathcal{N}([\![\Gamma]\!], [\![A]\!]) \to \Gamma \vdash A$, we obtain completeness of the deductive theory with respect to the class of models that the respective NbE model belongs to. Given the NbE models constructed in Subsections 3.1.3 and 3.2.3 this means that the deductive theories of λ_{IK} and λ_{IS4} (cf. Figs. 2 and 5) are (sound and) complete with respect to the class of possible-world models with an arbitrary frame and a reflexive–transitive frame, respectively.

Theorem 10. *Let $\Gamma : \mathit{Ctx}$ be a context and $A : \mathit{Ty}$ a type. If for all possible-world models $\mathcal{M}$ it is the case that $\mathcal{M}([\![\Gamma]\!], [\![A]\!])$ is inhabited then there is a term $t : \Gamma \vdash A$ of λ_{IK}.*

Proof. Let $\mathcal{M}_0$ be the model we constructed in Subsection 3.1.3. Since $\mathcal{M}_0$ is a possible-world model, by assumption we have a morphism $p : \mathcal{M}_0(\llbracket \Gamma \rrbracket, \llbracket A \rrbracket)$. And lastly, since $\mathcal{M}_0$ is an NbE model, we have the term $\mathit{quote}\, p : \Gamma \vdash A$. □

Theorem 11. *Let $\Gamma : \mathit{Ctx}$ be a context and $A : \mathit{Ty}$ a type. If for all possible-world models $\mathcal{M}$ with a reflexive–transitive frame it is the case that $\mathcal{M}(\llbracket \Gamma \rrbracket, \llbracket A \rrbracket)$ is inhabited then there is a term $t : \Gamma \vdash A$ of λ_{IS4}.*

Proof. As for Theorem 10. □

Note that the proofs of Theorems 10 and 11 are constructive.

Decidability of the Equational Theory. As a corollary of the completeness and adequacy of an NbE model $\mathcal{N}$, i.e. $\Gamma \vdash t \sim u : A$ if and only if $\llbracket t \rrbracket = \llbracket u \rrbracket : \mathcal{N}(\llbracket \Gamma \rrbracket, \llbracket A \rrbracket)$, we obtain decidability of the equational theory from decidability of the equality of normal forms $n, m : \Gamma \vdash_{\mathrm{NF}} A$. Given the NbE models constructed in Subsections 3.1.3 and 3.2.3 this means that the equational theories of λ_{IK} and λ_{IS4} (cf. Fig. 7) are decidable.

To show that any of the following decision problems $P(x)$ is decidable we give a *constructive* proof of the proposition $\forall x.\, P(x) \vee \neg P(x)$. Such a proof can be understood as the construction of an algorithm d that takes as input an x and produces as output a Boolean $d(x)$, alongside a correctness proof that $d(x)$ is true if and only if $P(x)$ holds.

Theorem 12. *For any two terms $t, u : \Gamma \vdash A$ of λ_{IK} the problem whether $t \sim u$ is decidable.*

Proof. We first observe that for any two normal forms $n, m : \Gamma \vdash_{\mathrm{NF}} A$ of λ_{IK} the problem whether $n = m$ is decidable by proving $\forall n, m.\, n = m \vee n \neq m$ constructively. All the cases of an simultaneous induction on $n, m : \Gamma \vdash_{\mathrm{NF}} A$ are immediate.

Let $\mathcal{N}$ be the NbE model we constructed in Subsection 3.1.3. Completeness and adequacy of $\mathcal{N}$ imply that we have $t \sim u$ if and only if $\mathit{norm}\, t = \mathit{norm}\, u$ for the function $\mathit{norm} : \Gamma \vdash A \to \Gamma \vdash_{\mathrm{NF}} A, t \mapsto \mathit{quote} \llbracket t \rrbracket$. Now, $t \sim u$ is decidable because $\mathit{norm}\, t = \mathit{norm}\, u$ is decidable by the observation we started with. □

Theorem 13. *For any two terms $t, u : \Gamma \vdash A$ of λ_{IS4} the problem whether $t \sim u$ is decidable.*

Proof. As for Theorem 12. □

Denecessitation. The last of the consequences of the NbE model constructions we record is of a less generic flavour than the other three, namely it is an application of normal forms to a basic proof-theoretic result in modal logic.

Using invariance of truth in possible-world models under bisimulation[2] it can be shown that $\Box A$ is a valid formula of IK (or IS4) if and only if A is. A completeness theorem then implies the same for provability of $\Box A$ and A. The statement for proofs in λ_{IK} (and λ_{IS4}) can also be shown by inspection of normal forms as follows.

[2]Invariance of truth under bisimulation says that if w and v are two bisimilar worlds in two possible-world models $\mathcal{M}_0$ and $\mathcal{M}_1$, respectively, then for all formulas A it is the case that $\llbracket A \rrbracket_w$ holds in $\mathcal{M}_0$ if and only if $\llbracket A \rrbracket_v$ does in $\mathcal{M}_1$.

Firstly, we note that while deduction is not closed under arbitrary context extensions (including locks) it is closed under extensions (including locks) *on the left*:

Lemma 14 (cf. Clouston [12, Lemma A.1]). *Let $\Delta, \Gamma : Ctx$ be arbitrary contexts, both possibly containing locks, and $A : Ty$ an arbitrary type. There is an operation $\Gamma \vdash A \to \Delta, \Gamma \vdash A$ on terms of λ_{IK} (and λ_{IS4}), where Δ, Γ denotes context concatenation.*

Proof. By recursion on terms. □

And, secondly, we note that also a converse of this lemma holds by inspection of normal forms:

Lemma 15. *Let $\Delta, \Gamma : Ctx$ be arbitrary contexts, both possibly containing locks, $A : Ty$ an arbitrary type and $t : \Delta, \Gamma \vdash A$ a term of λ_{IK} (or λ_{IS4}) in the concatenated context Δ, Γ that does* not *mention any variables from Δ, then there is a term $t' : \Gamma \vdash A$ of λ_{IK} (or λ_{IS4}, respectively).*

Proof. Since normalization (see Theorems 4 and 7) does not introduce new free variables it suffices to prove the statement for terms in normal form. We do so by induction on normal forms $n : \Delta, \Gamma \vdash_{\mathrm{NF}} A$ (see Fig. 8). The only nonimmediate step is for n of the form $\mathsf{unbox}\, n'\, e$ for some neutral element $n' : \Delta' \vdash_{\mathrm{NE}} \Box A$ and $\Delta' \lhd \Delta \leq \Delta, \Gamma$. But in that case the induction hypothesis says that we have a neutral element $n'' : \cdot \vdash_{\mathrm{NE}} \Box A$, which is impossible. □

Note that some form of normalization seems to be needed in the proof of Lemma 15. More specifically, the "strengthening" of a term of the form $\mathsf{unbox}\, t\, e$ from the context $\cdot, 🔒, \cdot$ to the empty context $\cdot$ cannot possibly result in a term of the form $\mathsf{unbox}\, t'\, e'$ because there is *no* context Γ such that $\Gamma \lhd \cdot$ in λ_{IK}. As an example, consider the term $\mathsf{unbox}\,(\mathsf{box}\,(\lambda\, x.\, x))\,\mathsf{nil}$, which needs to be strengthened to $\lambda\, x.\, x$.

With these two lemmas at hand we are ready to prove denecessitation through normalization:

Theorem 16. *Let $A : Ty$ be an arbitrary type. There is a term $t : \cdot \vdash A$ of λ_{IK} (or λ_{IS4}) if and only if there is a term $u : \cdot \vdash \Box A$ of λ_{IK} (or λ_{IS4}, respectively), where $\cdot : Ctx$ denotes the* empty *context.*

Proof. From a term $t : \cdot \vdash A$ we can construct a term $t' : \cdot, 🔒 \vdash A$ using Lemma 14 and thus the term $u = \mathsf{box}\, t' : \cdot \vdash \Box A$.

In the other direction, from a term $u : \cdot \vdash \Box A$ we obtain a normal form $u' = \mathit{norm}\, u : \cdot \vdash_{\mathrm{NF}} \Box A$ using Theorems 4 and 7. By inspection of normal forms (see Fig. 8) we know that u' must be of the form $\mathsf{box}\, v$ for some normal form $v : \cdot, 🔒 \vdash_{\mathrm{NF}} A$, from which we obtain a term $t : \cdot \vdash A$ using Lemma 15 since the context $\cdot, 🔒$ does not declare any variables that could have been mentioned in v. □

This concludes this section on some consequences of the model constructions presented in this paper. Note that the consequences we recorded are completely independent of the concrete model construction. To wit, the two completeness theorems follow from the mere existence of an NbE model, and the decidability and denecessitation theorems follow from the mere existence of a normalization function.

5 PROGRAMMING-LANGUAGE APPLICATIONS

In this section, we discuss some implications of normalization for Fitch-style calculi for specific interpretations of the necessity modality in the context of programming languages. In particular, we show how normalization can be used to prove properties about program calculi by leveraging the shape of normal forms of terms. We extend the term calculi presented earlier with application-specific primitives, ensure that the extended calculi are in fact normalizing, and then use this result to prove properties such as capability safety, noninterference, and binding-time correctness. Note that we do not mechanize these results in Agda and do not prove these properties in their full generality, but only illustrate special cases. Although possible, proving the general properties requires further technical development that obscures the main idea underlying the use of normal forms for simplifying these proofs.

5.1 Capability Safety

Choudhury and Krishnaswami [11] present a modal type system based on IS4 for a programming language with implicit effects in the style of ML [30] and the computational lambda calculus [32]. In this language, programs need access to capabilities to perform effects. For instance, a primitive for printing a string requires a capability as an argument in addition to the string to be printed. Crucially, capabilities cannot be introduced within the language, and must be obtained either from the global context (called *ambient* capabilities) or as a function argument.

Let us denote the type of capabilities by Cap. Passing a printing capability c to a function of type $\mathsf{Cap} \Rightarrow \mathsf{Unit}$ in a language that uses capabilities to print yields a program that either (1) does not print, (2) prints using only the capability c, or (3) prints using ambient capabilities (and possibly c). A program that at most uses the capabilities that it is passed explicitly, as in the cases 1 and 2, is said to be *capability safe*. To identify such programs, Choudhury and Krishnaswami [11] introduce a comonadic modality $\Box$ to capture capability safety. Their type system is loosely based on the dual-context calculus for IS4 [34, 27]. A term of type $\Box A$ is enforced to be capability safe by making the introduction rule for $\Box$ "brutally" remove all capabilities from the typing context. As a result, programs with the type $\Box(\mathsf{Cap} \Rightarrow \mathsf{Unit})$ are denied ambient capabilities and thus guaranteed to behave like the cases 1 and 2.

Choudhury and Krishnaswami [11] characterize capability safety precisely using their *capability space* model. A capability space (X, w_X) is a set X and a weight relation w_X that assigns sets of capabilities to every member in X. In this model, they define a comonad that restricts the underlying set of a capability space to those elements that are only related to the empty set of capabilities. This comonad has a left adjoint that replaces the weight relation of a capability space by the relation that relates every element to the empty set of capabilities. This adjunction suggests that capability spaces are a model of λ_{IS4} and we may thus use λ_{IS4} to write programs that support reasoning about capability safety.

In this subsection, we present a calculus λ_{IS4}+Moggi$^{\mathrm{Cap}}$ that extends λ_{IS4} with a capability type and a monad for printing effects. We extend the normalization algorithm for λ_{IS4} to λ_{IS4}+Moggi$^{\mathrm{Cap}}$ and show that the resulting normal forms can be used to prove a kind of capability safety. In contrast to the language presented

by Choudhury and Krishnaswami [11], λ_{IS4}+Moggi^{Cap} models a language where effects are explicit in the type of a term. Languages with explicit effects, such as Haskell [7] (with the IO monad) or PureScript [19] (with the **Effect** monad), can also benefit from a mechanism for capability safety, and we begin with an example in a hypothetical extension of PureScript to illustrate this.

Example in PureScript. Let us consider web development in PureScript. A web application may consist of a mashup of several components, e.g. social media, news feed, or chat, provided by untrusted sources. A component is a function of type

```
type Component = Element -> Effect Unit
```

that takes as a parameter the DOM element where the component will be rendered. For the correct functioning of the web application, it is important that components do not interfere with each other in malicious ways. For example, a malicious component (of Bob) could illegitimately overwrite a DOM element (of Alice):

```
evilBob :: Component
evilBob e = do w      <- window
               doc    <- document w
               aliceE <- getElementById "alice.app" doc
               setTextContent "Alice has been hacked!" aliceE
```

The issue here is that Bob has unrestricted access to the function `window :: Effect Window`, and is able to obtain the DOM using `document :: Window -> Effect DOM` and overwrite an element that belongs to Alice. Capabilities can be leveraged to restrict the access to `window`. We can achieve this by extending PureScript with a type **WindowCap**, a type constructor **Box** that works similarly to Choudhury and Krishnaswami's $\Box$, and replacing the function `window` with a function `window' :: WindowCap -> Effect Window` that requires an additional capability argument. By making **WindowCap** an ambient capability that is available globally, all existing programs retain their unrestricted access to retrieve a window as before. The difference now, however, is that we can selectively restrict some programs and limit their access to **WindowCap** using **Box**. We can define a variant of the type **Component** as:

```
type Component' = Box (Element -> Effect Unit)
```

By requiring Bob to write a component of the type **Component'**, we are ensured that Bob cannot overwrite an element that belongs to Alice. This is because the **Box** type constructor used to define **Component'** disallows access to all ambient capabilities (including **WindowCap**), and thus restricts Bob to only using the given **Element** argument. In particular, the program `evilBob` cannot be reproduced with the type **Component'** since the substitute function `window'` requires a capability that is neither available as an argument nor as an ambient capability.

Extension with a Capability and a Monad. We extend λ_{IS4} with a monad for printing based on Moggi's monadic metalanguage [33]. We introduce a type $\mathsf{T}\,A$ that denotes a monadic computation that can print before returning a value of type A, a type Cap for capabilities, and a type String for strings. Fig. 13 summarizes the terms that correspond to this extension. The term construct print is used for printing. The equational

$$\textit{Ty} \quad A, B ::= \ldots \mid \mathsf{T}\,A \mid \mathsf{Cap} \mid \mathsf{String} \mid \mathsf{Unit} \qquad \textit{Ctx} \quad \Gamma ::= \ldots$$

$$\text{T-Intro}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{return}\, t : \mathsf{T}\,A} \qquad \text{T-Elim}\ \frac{\Gamma \vdash t : \mathsf{T}\,A \qquad \Gamma, A \vdash u : \mathsf{T}\,B}{\Gamma \vdash \mathsf{let}\, t\, u : \mathsf{T}\,B}$$

$$\text{Unit-Intro}\ \Gamma \vdash \mathsf{unit} : \mathsf{Unit} \qquad \text{String-Lit}\ \frac{}{\Gamma \vdash \mathsf{str}_s : \mathsf{String}}\ s \in \textit{String}$$

$$\text{T-Print}\ \frac{\Gamma \vdash c : \mathsf{Cap} \qquad \Gamma \vdash s : \mathsf{String}}{\Gamma \vdash \mathsf{print}\, c\, s : \mathsf{T}\,\mathsf{Unit}}$$

Fig. 13. Types, contexts and terms of λ_{IS4}+Moggi$^{\mathsf{Cap}}$ (omitting the unchanged rules of Figs. 5 and 11)

$$\text{T-}\beta\ \frac{\Gamma \vdash t : A \qquad \Gamma, A \vdash u : \mathsf{T}\,B}{\Gamma \vdash \mathsf{let}\,(\mathsf{return}\, t)\, u \sim \textit{subst}\,(\mathsf{ext}\,\mathsf{id}_s\, t)\, u} \qquad \text{T-}\eta\ \frac{\Gamma \vdash t : \mathsf{T}\,A}{\Gamma \vdash t \sim \mathsf{let}\, t\,(\mathsf{return}\,(\mathsf{var}\,\mathsf{zero}))}$$

$$\text{T-}\gamma\ \frac{\Gamma \vdash t_1 : A \qquad \Gamma, A \vdash t_2 : \mathsf{T}\,B \qquad \Gamma, B \vdash t_3 : \mathsf{T}\,C}{\Gamma \vdash \mathsf{let}\,(\mathsf{let}\, t_1\, t_2)\, t_3 \sim \mathsf{let}\, t_1\,(\mathsf{let}\, t_2\,(\mathit{wk}\,(\mathsf{keep}\,(\mathsf{drop}\,\mathsf{id}_{\leq}))\, t_3))}$$

Fig. 14. Equational theory for λ_{IS4}+Moggi$^{\mathsf{Cap}}$ (omitting the unchanged rules of Figs. 7 and 12)

theory of λ_{IS4}+Moggi$^{\mathsf{Cap}}$ and the corresponding normal forms are summarized in Fig. 14 and Fig. 15, respectively.

To extend the NbE model of λ_{IS4} with an interpretation for the monad, we use the standard techniques used for normalizing computational effects [4, 17]. The interpretation of the other primitive types also follows a standard pursuit [42]: we interpret Cap by neutrals of type Cap and String by the disjoint union of *String* and neutrals of type String. The difference in their interpretation is caused by the fact that there is no introduction form for the type Cap.

Proving Capability Safety. Programs that lack access to capabilities are necessarily capability safe. We say that a program $\Gamma \vdash p : A$ is *trivially capability safe* if there is a program $\cdot \vdash p' : A$ such that $\Gamma \vdash p \sim \mathit{leftConcat}_\Gamma\, p' : A$, where $\mathit{leftConcat}_\Gamma : \forall \Delta, A.\, \Delta \vdash A \to \Gamma, \Delta \vdash A$ can be defined similarly to the operation given by Lemma 14 for λ_{IS4}.

First, we prove an auxiliary lemma about normal forms with a capability in context.

Lemma 17. *For any context* Γ*, type* A *and normal form* $c : \mathsf{Cap}, \locked, \Gamma \vdash_{NF} n : A$ *there is a normal form* $\cdot, \locked, \Gamma \vdash_{NF} n' : A$ *such that* $n = \mathit{leftConcat}_{c:\mathsf{Cap}}\, n'$.

$$\textsc{Nf/T-Intro}\ \frac{\Gamma \vdash_{\text{NF}} m : A}{\Gamma \vdash_{\text{NF}} \mathsf{return}\, m : \mathsf{T}\, A} \qquad \textsc{Nf/T-Elim}\ \frac{\Gamma \vdash_{\text{NE}} n : \mathsf{T}\, A \qquad \Gamma, A \vdash_{\text{NF}} m : \mathsf{T}\, B}{\Gamma \vdash_{\text{NF}} \mathsf{let}\, n\, m : \mathsf{T}\, B}$$

$$\textsc{Nf/Unit-Intro}\ \Gamma \vdash_{\text{NF}} \mathsf{unit} : \mathsf{Unit} \qquad \textsc{Nf/Up-Cap}\ \frac{\Gamma \vdash_{\text{NE}} n : \mathsf{Cap}}{\Gamma \vdash_{\text{NF}} \mathsf{up}\, n : \mathsf{Cap}} \qquad \textsc{Nf/Up-String}\ \frac{\Gamma \vdash_{\text{NE}} n : \mathsf{String}}{\Gamma \vdash_{\text{NF}} \mathsf{up}\, n : \mathsf{String}}$$

$$\textsc{Nf/String-Lit}\ \frac{}{\Gamma \vdash_{\text{NF}} \mathsf{str}_s : \mathsf{String}}\ s \in \mathit{String}$$

$$\textsc{Nf/T-Print}\ \frac{\Gamma \vdash_{\text{NF}} c : \mathsf{Cap} \quad \Gamma \vdash_{\text{NF}} s : \mathsf{String} \quad \Gamma, \mathsf{Unit} \vdash_{\text{NF}} m : \mathsf{T}\, A}{\Gamma \vdash_{\text{NF}} \mathsf{let}\, (\mathsf{print}\, c\, s)\, m : \mathsf{T}\, A}$$

Fig. 15. Normal forms of λ_{IS4}+Moggi$^{\text{Cap}}$ (omitting the unchanged normal forms of λ_{IS4})

Proof. We prove the statement for both normal forms and neutral elements by mutual induction. The only nonimmediate case is when the neutral is of the form $c : \mathsf{Cap}, 🔒, \Gamma \vdash_{\text{NE}} \mathsf{unbox}\, n\, e : A$ for some $n : \Delta \vdash_{\text{NE}} \Box A$ and $e : \Delta \lhd_{\lambda_{\text{IS4}}} c : \mathsf{Cap}, 🔒, \Gamma$. We observe that there are no neutral elements of type $\Box A$ in context $c : \mathsf{Cap}$ and that hence Δ must contain the leftmost lock in $c : \mathsf{Cap}, 🔒, \Gamma$. Thus, this case also holds by induction hypothesis. □

Now, we observe that all terms $c : \mathsf{Cap} \vdash t : \Box A$ are trivially capability safe. By normalization, we have that $c : \mathsf{Cap} \vdash t \sim \mathit{norm}\, t : \Box A$. Given the definition of normal forms of λ_{IS4}+Moggi$^{\text{Cap}}$, $\mathit{norm}\, t$ must be box n for some normal form $c : \mathsf{Cap}, 🔒 \vdash_{\text{NF}} n : A$. By Lemma 17, there is a normal form $🔒 \vdash_{\text{NF}} n' : A$ such that $n = \mathit{leftConcat}_{\cdot,\mathsf{Cap}}\, n'$. Since the operation *leftConcat* commutes with box, i.e. $\mathit{leftConcat}_{\cdot,\mathsf{Cap}}\, (\mathsf{box}\, n') = \mathsf{box}\, (\mathit{leftConcat}_{\cdot,\mathsf{Cap}}\, n')$, we also have that $t \sim \mathsf{box}\, n = \mathit{leftConcat}_{\cdot,\mathsf{Cap}}\, (\mathsf{box}\, n')$. As a result, t must be trivially capability safe.

A consequence of this observation is that any term $c : \mathsf{Cap} \vdash t : \Box(\mathsf{T}\,\mathsf{Unit})$ is trivially capability safe. This means that t does not print since it could not possibly do so without a capability. Going further, we can also observe that $t \sim \mathsf{box}\, (\mathsf{return}\, \mathsf{unit}) : \Box(\mathsf{T}\,\mathsf{Unit})$, since the only normal form of type $\mathsf{T}\,\mathsf{Unit}$ in the empty context is $\cdot \vdash_{\text{NF}} \mathsf{return}\, \mathsf{unit} : \mathsf{T}\,\mathsf{Unit}$. Note that this argument (and the one above) readily adapts to a vector of capabilities $\vec{c}$ in context as opposed to a single capability c.

5.2 Information-Flow Control

Information-flow control (IFC) [37] is a technique used to protect the confidentiality of data in a program by tracking the flow of information within the program.

In type-based *static* IFC [e.g. 1, 38, 36] types are used to associate values with confidentiality levels such as *secret* or *public*. The type system ensures that secret

inputs do not interfere with public outputs, enforcing a security policy that is typically formalized as a kind of *noninterference* property [20].

Noninterference is proved by reasoning about the semantic behaviour of a program. Tomé Cortiñas and Valliappan [40] present a proof technique that uses normalization for showing noninterference for a static IFC calculus based on Moggi's monadic metalanguage [33]. This technique exploits the insight that normal forms represent equivalence classes of terms identified by their semantics, and thus reasoning about normal forms of terms (as opposed to terms themselves) vastly reduces the set of programs that we must take into consideration. Having developed normalization for Fitch-style calculi, we can leverage this technique to prove noninterference.

In this subsection, we extend λ_{IK} with Booleans (denoted λ_{IK}+Bool), extend the NbE model of λ_{IK} to λ_{IK}+Bool, and illustrate the technique of Tomé Cortiñas and Valliappan on λ_{IK}+Bool for proving noninterference. We interpret the type $\Box A$ as a secret of type A, and other types as public.

Extension with Booleans. Noninterference can be better appreciated in the presence of a type whose values are distinguishable by an external observer. To this extent, we extend λ_{IK} with a type Bool and corresponding introduction and elimination forms—as described in Fig. 16.

$$\textit{Ty} \quad A, B ::= \ldots \mid \mathsf{Bool} \qquad\qquad \textit{Ctx} \quad \Gamma ::= \ldots$$

$$\textsc{Bool-Intro-true}\ \ \Gamma \vdash \mathsf{true} : \mathsf{Bool} \qquad\qquad \textsc{Bool-Intro-false}\ \ \Gamma \vdash \mathsf{false} : \mathsf{Bool}$$

$$\textsc{Bool-Elim}\ \ \frac{\Gamma \vdash b : \mathsf{Bool} \qquad \Gamma, \Gamma' \vdash t_1 : A \qquad \Gamma, \Gamma' \vdash t_2 : A}{\Gamma, \Gamma' \vdash \mathsf{ifte}\ b\ t_1\ t_2 : A}$$

Fig. 16. Types, contexts and intrinsically-typed terms of λ_{IK}+Bool (omitting the unchanged rules of Fig. 5)

We modify the usual elimination rule for Bool by allowing the context of the conclusion ifte $b\ t_1\ t_2$ and branches t_1 and t_2 in the rule Bool-Elim to extend the context of the scrutinee b. This modification (following Clouston [12, Fig. 2]) enables the following *commuting conversion*, which is required to ensure that terms can be fully normalized and normal forms enjoy the subformula property:

$$\frac{\Delta \vdash b : \mathsf{Bool} \qquad \Delta, \Delta' \vdash t_1 : \Box A \qquad \Delta, \Delta' \vdash t_2 : \Box A \qquad e : \Delta, \Delta' \lhd \Gamma}{\Gamma \vdash \mathsf{unbox}\ (\mathsf{ifte}\ b\ t_1\ t_2)\ e \sim \mathsf{ifte}\ b\ (\mathsf{unbox}\ t_1\ e)\ (\mathsf{unbox}\ t_2\ e)}$$

A commuting conversion is required as usual for every other elimination rule, including the rule $\Rightarrow$-Elim. These are however standard and thus omitted here.

We extend the equational theory of λ_{IK} to λ_{IK}+Bool by adding the usual rules:

- ifte true $t_1\ t_2 \sim t_1$
- ifte false $t_1\ t_2 \sim t_2$
- $t \sim$ ifte t true false, for terms t of type Bool

The normal forms of λ_{IK}+Bool include those of λ_{IK} in addition to the following.

$$\textsc{Nf/Bool-Intro-true}\quad \Gamma \vdash_{\text{NF}} \mathsf{true} : \mathsf{Bool} \qquad \textsc{Nf/Bool-Intro-false}\quad \Gamma \vdash_{\text{NF}} \mathsf{false} : \mathsf{Bool}$$

$$\textsc{Nf/Bool-Elim}\quad \frac{\Gamma \vdash_{\text{NE}} n : \mathsf{Bool} \quad \Gamma, \Gamma' \vdash_{\text{NF}} m_1 : A \quad \Gamma, \Gamma' \vdash_{\text{NF}} m_2 : A}{\Gamma, \Gamma' \vdash_{\text{NF}} \mathsf{ifte}\, n\, m_1\, m_2 : A}$$

Observe that a neutral of type Bool is not immediately in normal form, and must be expanded as $\mathsf{ifte}\, n\, \mathsf{true}\, \mathsf{false}$. This is unlike neutrals of the type ι, which are in normal form by Rule Nf/Up.

To extend the NbE model of λ_{IK} with Booleans, we leverage the interpretation of sum types used by Abel and Sattler [2], who attribute their idea to Altenkirch and Uustalu [5]. This interpretation readily supports commuting conversions, and a minor refinement that reflects the change to the rule Bool-Elim yields a reifiable interpretation for Booleans in λ_{IK}+Bool.

Proving Noninterference. A program $\cdot \vdash f : \Box A \Rightarrow \mathsf{Bool}$ is *noninterferent* if it is the case that $\cdot \vdash \mathsf{app}\, f\, s_1 \sim \mathsf{app}\, f\, s_2 : \mathsf{Bool}$ for any two secrets $\cdot \vdash s_1, s_2 : \Box A$. By instantiating A to Bool, we can show that any program $\cdot \vdash f : \Box\mathsf{Bool} \Rightarrow \mathsf{Bool}$ is noninterferent and thus cannot leak a secret Boolean argument. In λ_{IK}+Bool, the type system ensures that data of type $\Box A$ type can only influence (or *flow* to) data of type $\Box B$, thus all programs of type $\Box\mathsf{Bool} \Rightarrow \mathsf{Bool}$ must be noninterferent. To show this, we analyze the possible normal forms of f and observe that they must be equivalent to a constant function, such as $\lambda\, x.\, \mathsf{true}$ or $\lambda\, x.\, \mathsf{false}$, which evidently does not use its input argument x and is thus noninterferent.

In detail, normal forms of type $\Box\mathsf{Bool} \Rightarrow \mathsf{Bool}$ must have the shape $\lambda\, x.\, m$, for some normal form $\cdot, \Box\mathsf{Bool} \vdash_{\text{NF}} m : \mathsf{Bool}$. If m is either true or false, then $\lambda\, x.\, m$ must be a constant function and we are done. Otherwise, it must be some normal form $\cdot, \Box\mathsf{Bool} \vdash_{\text{NF}} \mathsf{ifte}\, n\, m_1\, m_2 : \mathsf{Bool}$ with a neutral $n : \mathsf{Bool}$ either in context $\cdot$ or in context $\cdot, \Box\mathsf{Bool}$. Such a neutral could either be of shape $\mathsf{unbox}\, n'$ or $\mathsf{app}\, n''\, m'$ for some neutrals n' and n''. However, this is impossible, since the context of the neutral $\mathsf{unbox}\, n'$ must contain a lock, and neither the context $\cdot$ nor the context $\cdot, \Box\mathsf{Bool}$ do. The existence of n'' can also be similarly dismissed by appealing to the definition of neutrals.

Discussion. Observe that not all Fitch-style calculi are well-suited for interpreting the type $\Box A$ as a secret, because noninterference might not hold. In λ_{IS4}, the term $\lambda\, x.\, \mathsf{unbox}\, x : \Box A \Rightarrow A$ (axiom T) is well-typed but leaks the secret x, thus breaking noninterference. The validity of the interpretation of $\Box A$ as a secret depends on the calculus under consideration and the axioms it exhibits.

5.3 Partial Evaluation

Davies and Pfenning [14, 15] present a modal type system for staged computation based on IS4. In their system, the type $\Box A$ represents *code* of type A that is to be executed at a later stage, and the axioms of IS4 correspond to operations that manipulate code. The axiom $\mathsf{K} : \Box(A \Rightarrow B) \Rightarrow (\Box A \Rightarrow \Box B)$ corresponds to substituting

code in code, $\mathsf{T} : \Box A \Rightarrow A$ to evaluating code, and $4 : \Box A \Rightarrow \Box\Box A$ to further delaying the execution of code to a subsequent stage. A desired property of this type system is that code must only depend on code, and thus the term $\lambda\, x : A.\, \mathsf{box}\, x$ must be ill-typed.

Although λ_{IS4} exhibits the desired properties of a type system for staging, its equational theory in Fig. 12 does not reflect the semantics of staged computation. For example, the result of normalizing the term $\mathsf{box}\,(2 * \mathsf{unbox}\,(\mathsf{box}\,3))$ in λ_{IS4} extended with natural number literals and multiplication is $\mathsf{box}\,6$. While the result expected from reducing it in accordance with Davies and Pfenning's operational semantics is $\mathsf{box}\,(2 * 3)$. The equational theory of Fitch-style calculi in general do not take into account the occurrence of a term (such as the literal 3) under box, while this is crucial for Davies and Pfenning's semantics. We return to this discussion at the end of this subsection.

If we restrict our attention to a special case of staged computation in partial evaluation [26], however, the semantics of Fitch-style calculi are better suited. In the context of partial evaluation, the type $\Box A$ represents a *dynamic* computation of type A that must be executed at runtime, and other types represent *static* computations. Static and dynamic are also known as binding-time annotations, and they are used by a partial evaluator to evaluate all static computations.

In the term $\mathsf{box}\,(2 * \mathsf{unbox}\,(\mathsf{box}\,3))$, we consider the literal 3 to be annotated as dynamic since it occurs under box. The construct unbox strips this annotation and brings it back to static. The multiplication of static subterms 2 and $\mathsf{unbox}\,(\mathsf{box}\,3)$ is however considered annotated dynamic since it itself occurs under box. As a result, a partial evaluator that respects these annotations does not perform the multiplication and specializes the term to $\mathsf{box}\,(2 * 3)$—which matches the result of evaluating with Davies and Pfenning's staging semantics. Observe that the same partial evaluator would specialize the expression $2 * \mathsf{unbox}\,(\mathsf{box}\,3)$ to 6 since the multiplication does not occur under box and is thus considered to be annotated static.

The goal of a partial evaluator is to optimize runtime execution of a program by eagerly evaluating as many static computations as possible and yielding an optimal dynamic program. The term $\mathsf{box}\,6$ is more optimized than the term $\mathsf{box}\,(2 * 3)$ since the evidently static multiplication has also been evaluated. Normalization in a Fitch-style calculus yields the former result, and the gain in optimality can be seen as a form of *binding-time improvement* [26] that is performed automatically during normalization.

In this subsection, we extend λ_{IK} with natural number literals and multiplication (denoted λ_{IK}+N), and extend the NbE model of λ_{IK} to λ_{IK}+N. We use λ_{IK} as the base calculus since the other axioms are not needed in the context of partial evaluation [14, 15]. The resulting normalization function yields an optimal partial evaluator for λ_{IK}+N. In partial evaluation, as with staging in general, we desire that a term $\lambda\, x : \mathsf{N}.\, \mathsf{box}\, x$ be disallowed, since a runtime execution of a dynamic computation must not have a static dependency. While this term is already ill-typed in λ_{IK}+N, we prove a kind of binding-time correctness property for λ_{IK}+N that implies that *no* term equivalent to $\lambda\, x : \mathsf{N}.\, \mathsf{box}\, x$ can exist.

Extension with Natural Number Literals and Multiplication. We extend λ_{IK} with a type N, a construct lift for including natural number literals, and an operation $*$ for multiplying terms of type N—as described in Fig. 17.

$$\textit{Ty} \quad A, B ::= \ldots \mid \mathsf{N} \qquad\qquad \textit{Ctx} \quad \Gamma ::= \ldots$$

$$\frac{}{\Gamma \vdash \mathsf{lift}\, k : \mathsf{N}}\ \textsc{N-Lift}\ k \in \mathbb{N} \qquad\qquad \frac{\Gamma \vdash t_1 : \mathsf{N} \qquad \Gamma \vdash t_2 : \mathsf{N}}{\Gamma \vdash t_1 * t_2 : \mathsf{N}}\ \textsc{N-Mul}$$

Fig. 17. Types, contexts, intrinsically-typed terms of λ_{IK}+N (omitting the unchanged rules of Fig. 5)

We extend the equational theory of λ_{IK} with some rules such as $\mathsf{lift}\, k_1 * \mathsf{lift}\, k_2 \sim \mathsf{lift}\, (k_1 * k_2)$ (for natural numbers k_1 and k_2), $\mathsf{lift}\, 0 * t \sim \mathsf{lift}\, 0$, $t \sim \mathsf{lift}\, 1 * t$, $t * \mathsf{lift}\, k \sim \mathsf{lift}\, k * t$, etc. The normal forms of λ_{IK}+N include those of λ_{IK} in addition to the following.

$$\frac{}{\Gamma \vdash_{\text{NF}} \mathsf{lift}\, 0 : \mathsf{N}}\ \textsc{N}_{\text{F}}/\textsc{N}_1 \qquad\qquad \frac{\Gamma \vdash_{\text{NE}} n_1 : \mathsf{N} \qquad \ldots \qquad \Gamma \vdash_{\text{NE}} n_j : \mathsf{N}}{\Gamma \vdash_{\text{NF}} \mathsf{lift}\, k * n_1 * \cdots * n_j : \mathsf{N}}\ \textsc{N}_{\text{F}}/\textsc{N}_2\ k \in \mathbb{N} \setminus \{0\}$$

The normal form $\mathsf{lift}\, k * n_1 * \cdots * n_j$ denotes a multiplication of a nonzero literal with a sequence of neutrals of type N, which can possibly be empty. The term $\mathsf{box}\, (2 * \mathsf{unbox}\, (\mathsf{box}\, 3))$ from earlier can be represented in λ_{IK}+N as $\mathsf{box}\, (\mathsf{lift}\, 2 * \mathsf{unbox}\, (\mathsf{box}\, (\mathsf{lift}\, 3)))$, and its normal form as $\mathsf{box}\, (\mathsf{lift}\, 6)$. To extend the NbE model for λ_{IK} to natural number literals and multiplication, we use the interpretation presented by Valliappan, Russo, and Lindley [42] for normalizing arithmetic expressions. Omitting the rule $\mathsf{lift}\, 0 * t \sim \mathsf{lift}\, 0$, this interpretation also resembles the one constructed systematically in the framework of Yallop, Glehn, and Kammar [43] for commutative monoids.

Proving Binding-Time Correctness. Binding-time correctness for a term $\cdot \vdash f : \mathsf{N} \Rightarrow \Box\mathsf{N}$ can be stated similar to noninterference: it must be the case that $\cdot \vdash \mathsf{app}\, f\, u_1 \sim \mathsf{app}\, f\, u_2 : \Box\mathsf{N}$ for any two arguments $\cdot \vdash u_1, u_2 : \mathsf{N}$. The satisfaction of this property implies that no well-typed term equivalent to $\lambda\, x : \mathsf{N}.\, \mathsf{box}\, x$ exists, since applying it to different arguments would yield different results. As before with noninterference, we can prove this property by case analysis on the possible normal forms of f. A normal form of f is either of the form $\lambda\, x.\, \mathsf{box}\, (\mathsf{lift}\, 0)$ or $\lambda\, x.\, \mathsf{box}\, (\mathsf{lift}\, k * n_1 * \cdots * n_j)$ for some natural number k and neutrals $n_1, \ldots, n_j$ of type N in context $\cdot, \mathsf{N}, \blacksquare$. In the former case, we are done immediately since $\lambda\, x.\, \mathsf{box}\, (\mathsf{lift}\, 0)$ is a constant function that evidently satisfies the desired criterion. In the latter case, we observe by induction that no such neutrals n_i exist, and hence f must be equivalent to the function $\lambda\, x.\, \mathsf{box}\, (\mathsf{lift}\, k)$, which is also constant.

As a part of binding-time correctness, we may also desire that nonconstant terms $\Box A \Rightarrow A$ like $\lambda\, x : \Box A.\, \mathsf{unbox}\, x$ be disallowed since a static computation must not have a dynamic dependency. This can also be shown by following an argument similar to the proof of noninterference in Subsection 5.2.

Discussion. The operational semantics for staged computation is given by Davies and Pfenning via translation to a dual-context calculus for IS4, where evaluation under the introduction rule box for $\Box$ is disallowed. While it is possible to implement a normalization function for λ_{IS4} that does not normalize under box, this then misses certain reductions that *are* enabled by the translation. For instance, the term box $(2 * \text{unbox}\,(\text{box}\,3))$ is already in normal form if we simply disallow normalization under box, while the translation ensures the reduction of unbox (box 3) by reducing the term to box $(2 * 3)$. This mismatch, in addition to the lack of a model for their system, makes the applicability of Fitch-style calculi for staged computation unclear.

6 RELATED AND FURTHER WORK

Fitch-Style Calculi. Fitch-style modal type systems [9, 29] adapt the proof methods of Fitch-style natural deduction systems for modal logic. In a Fitch-style natural deduction system, to eliminate a formula $\Box A$, we open a so-called strict subordinate proof and apply an "import" rule to produce a formula A. Fitch-style lambda calculi achieve a similar effect, for example in λ_{IK}, by adding a 🔒 to the context. To introduce a formula $\Box A$, on the other hand, we close a strict subordinate proof, and apply an "export" rule to a formula A—which corresponds to removing a 🔒 from the context. In the possible-world reading, adding a 🔒 corresponds to travelling to a future world, and removing it corresponds to returning to the original world.

The Fitch-style calculus λ_{IK} was presented for the logic IK by Borghuis [9] and Martini and Masini [29], and later investigated further by Clouston [12]. Clouston showed that 🔒 can be interpreted as the left adjoint of $\Box$, and proves a completeness result for a term calculus that extends λ_{IK} with a type former $\blacklozenge$ that internalizes 🔒. The extended term calculus is, however, somewhat unsatisfactory since the normal forms do not enjoy the subformula property. Normalization was also considered by Clouston, but only with Rule $\Box$-β and not Rule $\Box$-η. The normalization result presented here considers both rules, and the corresponding completeness result achieved using the NbE model does not require the extension of λ_{IK} with $\blacklozenge$. The decidability result that follows for the complete equational theory of λ_{IK} also appears to have been an open problem prior to our work.

For the logic IS4, there appear to be several possible formulations of a Fitch-style calculus, where the difference has to do with the definition of the rule $\lambda_{\text{IS4}}/\Box$-Elim. One possibility is to define unbox by explicitly recording the context extension as a part of the term former. Davies and Pfenning [14, 15] present such a system where they annotate the term former unbox as unbox_n to denote the number of 🔒s. Another possibility is to define unbox without any explicit annotations, thus leaving it ambiguous and to be inferred from a specific typing derivation. Such a system is presented by Clouston [12], and also discussed by Davies and Pfenning. In either formulation terms of type $\Box A \Rightarrow A$ (axiom T) and $\Box A \Rightarrow \Box\Box A$ (axiom 4) that satisfy the comonad laws are derivable. As a result, both formulations exhibit the logical equivalence $\Box\Box A \Leftrightarrow \Box A$. The primary difference lies in whether this logical equivalence can also be shown to be an isomorphism, i.e. whether the semantics of the modality $\Box$ is a comonad which is also *idempotent*. In Clouston's categorical semantics the modality $\Box$ is interpreted by an idempotent comonad. The λ_{IS4} calculus

presented here falls under the former category, where we record the extension explicitly using a premise instead of an annotation.

Gratzer, Sterling, and Birkedal [22] present yet another possibility that reformulates the system for IS4 in Clouston [12]. They further extend it with dependent types, and also prove a normalization result using NbE with respect to an equational theory that includes both Rule $\Box$-β and Rule $\Box$-η. Although their approach is semantic in the sense of using NbE, their semantic domain has a very syntactic flavour [22, Section 3.2] that obscures the elegant possible-world interpretation. For example, it is unclear as to how their NbE algorithm can be adapted to minor variations in the syntax such as in λ_{IK}, λ_{IK4} and λ_{IT}—a solution to which is at the very core of our pursuit. This difference also has to do with the fact that they are interested in NbE for type-checking (also called "untyped" or "defunctionalized" NbE), while we are interested in NbE for well-typed terms (and thus "typed" NbE), which is better suited for studying the underlying models. Furthermore, we also avoid several complications that arise in accommodating dependent types in a Fitch-style calculus, which is the main goal of their work.

Davies and Pfenning present their calculus for IS4 using a stack of contexts, which they call "Kripke-style", as opposed to the single Fitch-style context with a first-class delimiting operator 🔒. The elimination rule unbox_n for $\Box$ in the Kripke-style calculus for IS4 is indexed by an arbitrary natural number n specifying the number of stack frames the rule adds to the context stack of its premise. This index n corresponds to the modal accessibility premise of the Fitch-style unbox rule presented in Fig. 11. As in the Fitch-style presentation, Kripke-style calculi corresponding to the other logics IK, IT and IK4 can be recovered by restricting the natural numbers n for which the unbox_n rule is available. Hu and Pientka [24] present a normalization by evaluation proof for the Kripke-style calculi for all four logics IK, IT, IK4, and IS4. Their solution has a syntactic flavour similar to Gratzer, Sterling, and Birkedal [22] and also does not leverage the possible-world semantics. Furthermore, their proof is given for a single parametric system that encompasses the modal logics of interest, which need not be possible when we consider further modal axioms such as $\mathtt{R} : A \Rightarrow \Box A$.

Possible-World Semantics for Fitch-Style Calculi. Given that Fitch-style natural deduction for modal logic has itself been motivated by possible-world semantics, it is only natural that Fitch-style calculi can also be given possible-world semantics. It appears to be roughly understood that the 🔒 operator models some notion of a past world, but this has not been—to the best of our knowledge—made precise with a concrete definition that is supported by a soundness and completeness result. As noted earlier, this requires a minor refinement of the frame conditions that define possible-world models for intuitionistic modal logic given by Božić and Došen [10].

Dual-Context Calculi. Dual-context calculi [34, 14, 15, 27] provide an alternative approach to programming with the necessity modality using judgements of the form $\Delta; \Gamma \vdash A$ where Δ is thought of as the modal context and Γ as the usual (or "local") one. As opposed to a "direct" eliminator as in Fitch-style calculi, dual-context calculi feature a pattern-matching eliminator formulated as a let-construct. The let-construct allows a type $\Box A$ to be eliminated into an arbitrary type C, which induces an array of commuting conversions in the equational theory to attain normal

forms that obey the subformula property. Furthermore, the inclusion of an η-law for the $\Box$ type former complicates the ability to produce a unique normal form. Normalization (and, more specifically, NbE) for a pattern-matching eliminator—while certainly achievable—is a much more tedious endeavour, as evident from the work on normalizing sum types [6, 28, 2], which suffer from a similar problem. Presumably for this reason, none of the existing normalization results for dual-context calculi consider the η-law. The possible-world semantics of dual-context calculi is also less apparent, and it is unclear how NbE models can be constructed as instances of that semantics.

Multimodal Type Theory (MTT). Gratzer et al. [23] present a multimodal dependent type theory that for every choice of mode theory yields a dependent type theory with multiple interacting modalities. In contrast to Fitch-style calculi, their system features a variable rule that controls the use of variables of modal type in context. Further, the elimination rule for modal types is formulated in the style of the let-construct for dual-context calculi. In a recent result, Gratzer [21] proves normalization for multimodal type theory. In spite of the generality of multimodal type theory, it is worth noting that the normalization problem for Fitch-style calculi, when considering the full equational theory, is not a special case of normalization for multimodal type theory.

Further Modal Axioms. The possible-world semantics and NbE models presented here only consider the logics IK, IT, IK4 and IS4. We wonder if it would be possible to extend the ideas presented here to further modal axioms such as $\mathrm{R} : A \Rightarrow \Box A$ and $\mathrm{GL} : \Box(\Box A \Rightarrow A) \Rightarrow \Box A$, especially considering that the calculi may differ in more than just the elimination rule for the $\Box$ type.

DATA AVAILABILITY STATEMENT

The Agda mechanization [41] of the calculi λ_{IK} and λ_{IS4} and their normalization algorithms are available in the Zenodo repository.

ACKNOWLEDGMENTS

We would like to thank Andreas Abel, Thierry Coquand, and Graham Leigh for their feedback on earlier versions of this work. We would also like to thank the anonymous referees of both the paper and the artifact for their valuable comments and helpful suggestions.

This work is supported by the Swedish Foundation for Strategic Research (SSF) under the projects Octopi (Ref. RIT17-0023R) and WebSec (Ref. RIT17-0011).

REFERENCES

[1] Martín Abadi et al. "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 147–160. DOI: 10.1145/292540.292555. URL: https://doi.org/10.1145/292540.292555.

[2] Andreas Abel and Christian Sattler. "Normalization by Evaluation for Call-By-Push-Value and Polarized Lambda Calculus". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 2019, 3:1–3:12. DOI: 10.1145/3354166.3354168. URL: https://doi.org/10.1145/3354166.3354168.

[3] Andreas Abel et al., *Agda 2* version 2.6.2.1, 2005–2021. Chalmers University of Technology and Gothenburg University. LIC: BSD3. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php, VCS: https://github.com/agda/agda.

[4] Danel Ahman and Sam Staton. "Normalization by Evaluation and Algebraic Effects". In: *Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2013, New Orleans, LA, USA, June 23-25, 2013*. Ed. by Dexter Kozen and Michael W. Mislove. Vol. 298. Electronic Notes in Theoretical Computer Science. Elsevier, 2013, pp. 51–69. DOI: 10.1016/j.entcs.2013.09.007. URL: https://doi.org/10.1016/j.entcs.2013.09.007.

[5] Thorsten Altenkirch and Tarmo Uustalu. "Normalization by Evaluation for lambda^{-2}". In: *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings*. Ed. by Yukiyoshi Kameyama and Peter J. Stuckey. Vol. 2998. Lecture Notes in Computer Science. Springer, 2004, pp. 260–275. DOI: 10.1007/978-3-540-24754-8_19. URL: https://doi.org/10.1007/978-3-540-24754-8%5C_19.

[6] Thorsten Altenkirch et al. "Normalization by Evaluation for Typed Lambda Calculus with Coproducts". In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 2001, pp. 303–310. DOI: 10.1109/LICS.2001.932506. URL: https://doi.org/10.1109/LICS.2001.932506.

[7] Lennart Augustsson et al., *Haskell* 1990. URL: https://www.haskell.org/.

[8] Ulrich Berger and Helmut Schwichtenberg. "An Inverse of the Evaluation Functional for Typed lambda-calculus". In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, 1991, pp. 203–211. DOI: 10.1109/LICS.1991.151645. URL: https://doi.org/10.1109/LICS.1991.151645.

[9] Valentijn Anton Johan Borghuis. *Coming to terms with modal logic*. On the interpretation of modalities in typed λ-calculus, Dissertation, Technische Universiteit Eindhoven, Eindhoven, 1994. Technische Universiteit Eindhoven, Eindhoven, 1994, pp. x+219.

[10] Milan Božić and Kosta Došen. "Models for normal intuitionistic modal logics". In: *Studia Logica* 43.3 (1984), pp. 217–245. ISSN: 0039-3215. DOI: 10.1007/BF02429840. URL: https://doi.org/10.1007/BF02429840.

[11] Vikraman Choudhury and Neel Krishnaswami. "Recovering purity with comonads and capabilities". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 111:1–111:28. DOI: 10.1145/3408993. URL: https://doi.org/10.1145/3408993.

[12] Ranald Clouston. "Fitch-Style Modal Lambda Calculi". In: *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. Ed. by Christel Baier and Ugo Dal Lago. Vol. 10803. Lecture Notes in Computer Science. Springer, 2018, pp. 258–275. DOI: 10.1007/978-3-319-89366-2_14. URL: https://doi.org/10.1007/978-3-319-89366-2%5C_14.

[13] Catarina Coquand. "A Formalised Proof of the Soundness and Completeness of a Simply Typed Lambda-Calculus with Explicit Substitutions". In: *High. Order Symb. Comput.* 15.1 (2002), pp. 57–90. DOI: 10.1023/A:1019964114625. URL: https://doi.org/10.1023/A:1019964114625.

[14] Rowan Davies and Frank Pfenning. "A Modal Analysis of Staged Computation". In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. Ed. by Hans-Juergen Boehm and Guy L. Steele Jr. ACM Press, 1996, pp. 258–270. DOI: 10.1145/237721.237788. URL: https://doi.org/10.1145/237721.237788.

[15] Rowan Davies and Frank Pfenning. "A modal analysis of staged computation". In: *J. ACM* 48.3 (2001), pp. 555–604. DOI: 10.1145/382780.382785. URL: https://doi.org/10.1145/382780.382785.

[16] W. B. Ewald. "Intuitionistic Tense and Modal Logic". In: *J. Symb. Log.* 51.1 (1986), pp. 166–179. DOI: 10.2307/2273953. URL: https://doi.org/10.2307/2273953.

[17] Andrzej Filinski. "Normalization by Evaluation for the Computational Lambda-Calculus". In: *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*. Ed. by Samson Abramsky. Vol. 2044. Lecture Notes in Computer Science. Springer, 2001, pp. 151–165. DOI: 10.1007/3-540-45413-6_15. URL: https://doi.org/10.1007/3-540-45413-6%5C_15.

[18] Gisèle Fischer-Servi. "Semantics for a class of intuitionistic modal calculi". In: *Italian studies in the philosophy of science*. Vol. 47. Boston Stud. Philos. Sci. Reidel, Dordrecht-Boston, Mass., 1981, pp. 59–72.

[19] Phil Freeman, *PureScript* 2013. URL: https://www.purescript.org/.

[20] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014. URL: https://doi.org/10.1109/SP.1982.10014.

[21] Daniel Gratzer. "Normalization for multimodal type theory". In: *CoRR* abs/2106.01414 (2021). arXiv: 2106.01414. URL: https://arxiv.org/abs/2106.01414.

[22] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. "Implementing a modal dependent type theory". In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 107:1–107:29. DOI: 10.1145/3341711. URL: https://doi.org/10.1145/3341711.

[23] Daniel Gratzer et al. "Multimodal Dependent Type Theory". In: *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*. Ed. by Holger Hermanns et al. ACM, 2020, pp. 492–506. DOI: 10.1145/3373718.3394736. URL: https://doi.org/10.1145/3373718.3394736.

[24] Jason Z. S. Hu and Brigitte Pientka. "An Investigation of Kripke-style Modal Type Theories". In: *CoRR* abs/2206.07823 (2022). arXiv: 2206.07823. URL: https://arxiv.org/abs/2206.07823.

[25] C. Barry Jay and Neil Ghani. "The Virtues of Eta-Expansion". In: *J. Funct. Program.* 5.2 (1995), pp. 135–154. DOI: 10.1017/S0956796800001301. URL: https://doi.org/10.1017/S0956796800001301.

[26] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993. ISBN: 978-0-13-020249-9.

[27] G. A. Kavvos. "Dual-Context Calculi for Modal Logic". In: *Log. Methods Comput. Sci.* 16.3 (2020). URL: https://lmcs.episciences.org/6722.

[28] Sam Lindley. "Extensional Rewriting with Sums". In: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*. Ed. by Simona Ronchi Della Rocca. Vol. 4583. Lecture Notes in Computer Science. Springer, 2007, pp. 255–271. DOI: 10.1007/978-3-540-73228-0_19. URL: https://doi.org/10.1007/978-3-540-73228-0%5C_19.

[29] Simone Martini and Andrea Masini. "A computational interpretation of modal proofs". In: *Proof theory of modal logic (Hamburg, 1993)*. Vol. 2. Appl. Log. Ser. Kluwer Acad. Publ., Dordrecht, 1996, pp. 213–241. DOI: 10.1007/978-94-017-2798-3_12. URL: https://doi.org/10.1007/978-94-017-2798-3_12.

[30] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990. ISBN: 978-0-262-63132-7.

[31] Kenji Miyamoto and Atsushi Igarashi. "A modal foundation for secure information flow". In: *In Proceedings of IEEE Foundations of Computer Security (FCS)*. 2004, pp. 187–203.

[32] 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*. IEEE Computer Society, 1989, pp. 14–23. DOI: 10.1109/LICS.1989.39155. URL: https://doi.org/10.1109/LICS.1989.39155.

[33] Eugenio Moggi. "Notions of Computation and Monads". In: *Inf. Comput.* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4.

[34] Frank Pfenning and Rowan Davies. "A judgmental reconstruction of modal logic". In: *Math. Struct. Comput. Sci.* 11.4 (2001), pp. 511–540. DOI: 10.1017/S0960129501003322. URL: https://doi.org/10.1017/S0960129501003322.

[35] Gordon D. Plotkin and Colin Stirling. "A Framework for Intuitionistic Modal Logics". In: *Proceedings of the 1st Conference on Theoretical Aspects of Reasoning about Knowledge, Monterey, CA, USA, March 1986*. Ed. by Joseph Y. Halpern. Morgan Kaufmann, 1986, pp. 399–406.

[36] Alejandro Russo, Koen Claessen, and John Hughes. "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, 2008, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289.

[37] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security". In: *IEEE J. Sel. Areas Commun.* 21.1 (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121. URL: https://doi.org/10.1109/JSAC.2002.806121.

[38] Naokata Shikuma and Atsushi Igarashi. "Proving Noninterference by a Fully Complete Translation to the Simply Typed Lambda-Calculus". In: *Log. Methods Comput. Sci.* 4.3 (2008). DOI: 10.2168/LMCS-4(3:10)2008. URL: https://doi.org/10.2168/LMCS-4(3:10)2008.

[39] Alex K. Simpson. "The proof theory and semantics of intuitionistic modal logic". PhD thesis. University of Edinburgh, UK, 1994. URL: http://hdl.handle.net/1842/407.

[40] Carlos Tomé Cortiñas and Nachiappan Valliappan. "Simple Noninterference by Normalization". In: *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, CCS 2019, London, United Kingdom, November 11-15, 2019*. Ed. by Piotr Mardziel and Niki Vazou. ACM, 2019, pp. 61–72. DOI: 10.1145/3338504.3357342. URL: https://doi.org/10.1145/3338504.3357342.

[41] Nachiappan Valliappan, Fabian Ruch, and Carlos Tomé Cortiñas, *Artifact for "Normalization for Fitch-Style Modal Calculi"* version 1.1.0, Aug. 2022. LIC: CC-BY-4.0. DOI: 10.5281/zenodo.6957191, URL: https://doi.org/10.5281/zenodo.6957191.

[42] Nachiappan Valliappan, Alejandro Russo, and Sam Lindley. "Practical normalization by evaluation for EDSLs". In: *Haskell 2021: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell, Virtual Event, Korea, August 26-27, 2021*. Ed. by Jurriaan Hage. ACM, 2021, pp. 56–70. DOI: 10.1145/3471874.3472983. URL: https://doi.org/10.1145/3471874.3472983.

[43] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. "Partially-static data as free extension of algebras". In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 100:1–100:30. DOI: 10.1145/3236795. URL: https://doi.org/10.1145/3236795.

B

Practical Normalization by Evaluation for EDSLs

Abstract. Embedded domain-specific languages (eDSLs) are typically implemented in a rich host language, such as Haskell, using a combination of deep and shallow embedding techniques. While such a combination enables programmers to exploit the execution mechanism of Haskell to build and specialize eDSL programs, it blurs the distinction between the host language and the eDSL. As a consequence, extension with features such as sums and effects requires a significant amount of ingenuity from the eDSL designer. In this paper, we demonstrate that Normalization by Evaluation (NbE) provides a principled framework for building, extending, and customizing eDSLs. We present a comprehensive treatment of NbE for deeply embedded eDSLs in Haskell that involves a rich set of features such as sums, arrays, exceptions and state, while addressing practical concerns about normalization such as code expansion and the addition of domain-specific features.

1 INTRODUCTION

An embedded domain-specific language (eDSL) [24, 26] is a seamless implementation of a domain-specific language (DSL) as a library in a host language. Haskell is particularly well suited as a host for eDSLs as witnessed by the variety of practical Haskell eDSLs covering domains as diverse as circuit design [12], database querying [25], digital signal processing [6], graphics acceleration [14], and security [38]. Haskell eDSL developers have at their disposal all of Haskell's features such as higher-order functions, extensible syntax, and a rich type-system. It is common to represent programs in an eDSL using a data type that denotes them explicitly, together with compilers and interpreters that manipulate values of this type. Let us consider such a data type $Exp :: * \to *$ parameterized by the type of the expression it denotes. Whereas a value of type *Int* in Haskell denotes an integer *value*, a value of type *Exp Int* denotes an integer *expression*. (We use the words "program" and "expression" interchangeably in the rest of the paper.)

Often, eDSL designers face a choice between either adding complex features to an eDSL or keeping the core eDSL simple and exploiting the host language to construct programs. Should the eDSL support pairs in expressions ($Exp\ (a, b)$), or should it use pairs of expressions ($(Exp\ a, Exp\ b)$)? Should the eDSL support functions ($Exp\ (Int \to Int)$) directly or should it instead rely on Haskell functions ($Exp\ Int \to Exp\ Int$) to build programs? As the complexity increases, it can become difficult to draw a line between the end of the host language and the beginning of the eDSL.

In an eDSL program we may think of a value of type *Int* as a *static* integer that is known at compile-time, and a value of type *Exp Int* as a *dynamic* integer that is known only at runtime. The *stage separation* of values as static and dynamic corresponds to a manual form of *binding-time analysis* in partial evaluation [27], and presents an opportunity to exploit Haskell's execution mechanism to evaluate static computations in an eDSL program. In other words, *static values belong to the host language*, whereas *dynamic values belong to the eDSL*. For example, consider the following implementation of the exponentiation function that receives two integer arguments n and x and returns x^n

$$power_1 :: Int \to Exp\ Int \to Exp\ Int$$
$$power_1\ n\ x = \textbf{if}\ (n \leqslant 1)\ \textbf{then}\ x\ \textbf{else}\ (x * (power_1\ (n - 1))$$

where $(*) :: Exp\ Int \to Exp\ Int \to Exp\ Int$. The type of $power_1$ ensures the first argument is static, and using this function, we can evaluate the expression $power_1\ 3\ x$ for some $x :: Exp\ Int$ to generate the specialized expression $x * x * x$. Even though the definition of $power_1$ uses a conditional (**if** ... **then** ... **else**), comparison ($n \leqslant 1$) and function recursion ($power_1\ (n - 1)$), these have all been evaluated (by Haskell) and removed in the specialized expression.

Though separation of stages enables the programmer to manually specify those parts of an eDSL program that must be evaluated by Haskell, it also burdens them to maintain multiple variants of the same program. In addition to $power_1$, we may also demand the following variants of the exponentiation function, each corresponding to a different separation of stages for its arguments and result.

$power_0 :: Int \to Int \to Int$
$power_1 :: Int \to Exp\ Int \to Exp\ Int$
$power_2 :: Exp\ Int \to Int \to Exp\ Int$
$power_3 :: Int \to Int \to Exp\ Int$
$power_4 :: Exp\ Int \to Exp\ Int \to Exp\ Int$
$power_5 :: Int \to Exp\ (Int \to Int)$
$power_6 :: Exp\ Int \to Exp\ (Int \to Int)$
$power_7 :: Exp\ (Int \to Int \to Int)$

The need for multiple variants can be mitigated to some extent by using an overloading mechanism that automatically lifts *Int* to *Exp Int*, and converts back and forth between some static and dynamic representations, such as $Exp\ (a \to b)$ and $Exp\ a \to Exp\ b$. This is done, for example, in Feldspar [6]. However, conversion between representations does not work for types with multiple introduction forms such as sum types: we cannot convert an expression of type $Exp\ (Either\ a\ b)$ to $Either\ (Exp\ a)\ (Exp\ b)$ as the precise injection may not be known until runtime.

Normalization by Evaluation (NbE) [9], is a program specialization technique that offers a solution to this problem by making specialization automatic, *without the need for manual stage separation*. Using the NbE approach, all variants of the exponentiation function can be recovered from the implementation of $power_7 :: Exp\ (Int \to Int \to Int)$ depending on the availability of the arguments at the site of invocation, i.e., depending on how $power_7$ is used.

Unlike traditional normalization techniques, NbE bypasses rewriting entirely and instead normalizes an expression by evaluating it using a special interpreter. While NbE techniques for well-typed languages, also known as *typed NbE*, have found a number of theoretical applications such as deciding equivalence of lambda-calculus with sums [3], proving completeness [16], and coherence theorems [11], the practical relevance of typed NbE remains relatively less well-understood. This paper argues that typed NbE is particularly well-suited for specializing eDSL programs in Haskell given the natural reliance on a host language. Indeed, existing techniques for embedding DSLs in Haskell (e.g. the work of Svenningsson and Axelsson [40] on combining deep and shallow embeddings), which may at first seem somewhat ad hoc, can be viewed as instances of NbE.

The contributions of this paper are as follows.

- The first comprehensive practical treatment of NbE for eDSLs.
- A coherent combination of NbE techniques to deal with a rich set of features such as sums, arrays, exceptions, and state—and in particular—a detailed and extensible account of their interaction.
- Practical extensions of standard NbE techniques to implement a richer set of domain-specific equations, and variations that control unnecessary code expansion.
- Examples showing that NbE provides a principled alternative to ad hoc techniques that combine deep and shallow embedding to implement fusion for functions, loops and arrays in an eDSL.

The complete Haskell source code and examples in this paper can be found in the accompanying material available at https://github.com/nachivpn/nbe-edsl.

2 NORMALIZING EDSL PROGRAMS

This section showcases our implementation with examples of normalizing eDSL programs using NbE. We begin with standard examples of normalizing the exponentiation function and array operations, and then show examples that illustrate normalization of programs that contain side-effects, branching, and an intricate interaction between them.

Normalization is performed by a function $norm :: Rf\ a \Rightarrow Exp\ a \to Exp\ a$, and the result is observed by printing the resulting expression. The type class constraint *Rf* limits the type of an expression to the types recognized by the eDSL, and is defined along with the data type *Exp* in the next section. The name *Rf* is short for reifiable. For convenience, we do not program with the constructors of *Exp* directly, and instead use derived combinators and "smart constructors" that provide a programming interface to the eDSL. This means that the observed result of normalizing an eDSL program is that of its internal representation, and may not directly resemble the surface program.

We make use of a form of higher-order abstract syntax (HOAS) [34] in order to repurpose the binding features of Haskell in the eDSL. Thus the constructor that constructs an expression of a function type $Exp\ (a \to b)$ (*Lam* in Figure 2) takes a Haskell function on expressions $Exp\ a \to Exp\ b$ as its argument. Our focus here is on practical implementation so we do not concern ourselves with subtleties such as ruling out so called exotic terms or exotic types [5] in the internal representation of expressions. Nevertheless, it is a routine exercise to adapt our approach to use standard techniques to preclude such infelicities, for instance by using an abstract type to hide the concrete type constructors [36] or moving to a tagless representation [5, 13] whereby the smart constructors are first-class.

Normalizing exponentiation. Consider again the exponentiation function from the previous section, and suppose that it is implemented as follows.

```
power :: Exp (Int → Int → Int)
power = lam $ λn → lam $ λx → rec n (f x) 1
  where f x = lam $ λ_ → lam $ λacc → (x * acc)
```

This implementation corresponds to the $power_7$ variant, and is implemented using expression combinators: $lam :: (Exp\ a \to Exp\ b) \to Exp\ (a \to b)$ is a lambda expression combinator and $rec :: Exp\ Int \to Exp\ (Int \to a \to a) \to Exp\ a \to Exp\ a$ is a primitive recursion combinator such that $rec\ n\ g\ a$ is equivalent to $g\ 1\ (g\ 2\ (...(g\ n\ a)))$. Although possible, note that the type of *rec* is not entirely wrapped under *Exp* as $Exp\ (Int \to (Int \to a \to a) \to a \to a)$. This choice prevents unnecessary clutter caused by explicit function application in the expression language, and trades some specialization power (i.e., the subsumption of some stage separations) for a more convenient interface. We make this choice for all primitive combinators that require multiple arguments.

An expression of a function type can be applied using the combinator $app :: Exp\ (a \to b) \to Exp\ a \to Exp\ b$ as $app\ (power\ 3)$, where the argument is a numeral expression $3 :: Exp\ Int$. We can normalize this expression in the Haskell interpreter GHCi using the function *norm* as follows.

```
*NbE.OpenNbE> norm (app power 3)
λx.(x * (x * x))
```

The result is a pretty-printed representation of the expression syntax of the *Exp* data type—here a function that returns the cube of its argument. Observe that the result is slightly more optimal than that of a textbook partial evaluator that returns $\lambda x.(x*(x*(x*1)))$ by unrolling the recursion. This optimization is a simple instance of NbE's ability to aggressively reduce arithmetic expressions even in the presence of unknown values—we return to this in Section 6.

Note here that the specialization of *power* is automatic and there was no need to manually separate the stages of arguments as static (*Int*) and dynamic (*Exp Int*). We consider the entire expression to be dynamic, and leave it to the normalizer to identify the best specialization strategy.

As another example, consider normalizing an invocation of *power* with flipped arguments using a utility function *flip′*.

```
*NbE.OpenNbE> norm (app (flip′ power) 3)
λn.(Rec n (λy.λacc.(3 * acc)) 1)
```

Observe that the (expected) definition of *flip′* has been removed in the result, producing a more optimal function.

Normalizing array operations. Normalization can be used to achieve fusion of operations over arrays such as map and fold [33]. We consider immutable *pull arrays* [41] in our eDSL, and an expression of the array type is denoted by the type *Exp* (*Arr a*), where *a* denotes the type of the elements in the array. The map and fold operations are given by derived combinators, whose types and corresponding fusion laws are given as below.

```
mapArr :: Exp (a → b) → Exp (Arr a) → Exp (Arr b)
foldArr :: Exp (b → a → b) → Exp b → Exp (Arr a) → Exp b
  -- fusion laws:
  -- 1. mapArr f (mapArr g arr) = mapArr (f . g) arr
  -- 2. foldrArr f x (mapArr g arr) = foldArr (f . g) x arr
```

These combinators are derived using simpler expression constructors, that for e.g., create an array (*NewArr*), or perform recursion (*Rec*), and the fusion laws follow from the equations that specify their behavior.

Using these combinators, we may implement a function (expression) *mapMap* that maps twice over a given argument array, first with the function (+1), and then with (+2).

```
mapMap :: Exp (Arr Int → Arr Int)
mapMap = lam $ λarr →
  mapArr (lam (+2)) (mapArr (lam (+1)) arr)
```

By the first fusion law, this expression is equivalent to one that maps once with (+3) as: *mapArr* (*lam* (+3)) *arr*. Normalizing *mapMap* returns a new array which has the same length as the argument array *arr*, and whose elements are the elements of *arr* incremented by 3.

```
∗NbE.OpenNbE> norm mapMap
λarr.(NewArr (LenArr arr) (λi.(arr ! i) + 3))
```

The result is indeed the expression constructed by applying the derived combinator *mapArr* as *mapArr* (*lam* (+3)) *arr*. Besides map fusion, NbE also eliminates the function composition from the fused function (+2) ∘ (+1) and performs constant folding to obtain (+3).

To illustrate the second fusion law, consider the following function, *mapFold*, that first maps (+2) over a given array and then computes the sum of the result using *foldArr*.

```
mapFold :: Exp (Arr Int → Int)
mapFold = lam $ λarr → foldArr go 0 (mapArr (lam (+2)) arr)
  where go = lam $ λacc → lam $ λx → acc + x
```

By the second fusion law, this expression is equivalent to one which simply folds the entire array as: *foldArr* (*lam* (λ*acc* → *lam* (λ*x* → *acc* + *x* + 2))) 0 *arr*. Normalizing *mapFold* yields the following result.

```
∗NbE.OpenNbE> norm mapFold
λarr.(Rec (LenArr arr) (λi.λacc.acc + (arr ! i) + 2) 0)
```

The normalized function receives an argument array, and performs recursion over its length to compute the sum of its elements, each of which has been incremented by 2.

Normalizing branching programs. Branching programs, or programs that perform a case analysis over a value of a sum type, complicate normalization. The difficulty arises from the fact that the outcome of a case analysis over an unknown value cannot be determined at normalization time. NbE offers a modular solution to address this difficulty and achieve normalization for branching programs, as we shall see later in Section 4.

Consider the following branching program, *prgBr*, that illustrates a scenario where map fusion on arrays is interrupted by a case analysis on an unknown value.

```
prgBr :: Exp (Either Int Int → Arr Int → Arr Int)
prgBr = lam $ λscr → lam $ λarr →
  mapArr (lam (+1)) $ case′ scr
    (lam $ λx → mapArr (lam (+x)) arr)
    (lam $ λy → arr)
```

It performs a case analysis using the combinator *case′* :: *Exp* (*Either a b*) → *Exp* (*a* → *c*) → *Exp* (*b* → *c*) on the argument *scr* (an unknown value), and if the left injection is found with an integer *x*, it returns an array that increments elements of *arr* by *x*, else *arr* is returned as found otherwise. The array returned by *case′* is further incremented by 1.

Normalizing *prgBr* yields the following result.

```
∗NbE.OpenNbE> norm prgBr
(λscr.(λarr.NewArr (LenArr arr)
  (λi.Case scr (λx.((arr ! i) + x + 1)) (λy.((arr ! i) + 1)))))
```

The normalized function returns a new array whose elements are given by performing case analysis on *scr*. Observe that the effect of *mapArr* (*lam* (+1)) in *prgBr* has been fused with the application of *mapArr* in the first branch, and left unaltered in the second branch. The normalized program delays case analysis on *scr* to the point at which it is required, thus avoiding the materialisation of an intermediate array.

Normalizing stateful programs. Programs with side-effects can be also normalized using NbE, and the following example illustrates such a program that writes to and reads from a global state in a *State* monad.

```
prgSt :: Exp (Arr Int → State (Arr Int) Int)
prgSt = lam $ λarr →
  put (mapArr (lam (+2)) arr)
  ≫st put (mapArr (lam (+1)) arr
  ≫st get ≫=st (lam $ λarr′ → returnst (ixArr arr′ 0)))
```

The program *prgSt* receives an integer array *arr*, and returns an *Int* by writing to and reading from (using combinators *get* and *put*) a global state that contains an array of type *Arr Int*. Precisely, it performs the following actions (sequenced using monadic combinators $\gg_{st}$ and $\gg\!=_{st}$):

- writes the result of mapping over *arr* with (+2)
- writes the result of mapping over *arr* with (+1)
- reads the array from state, and returns its first element

The combinators *put*, *get*, $\gg_{st}$, $\gg\!=_{st}$ and $return_{st}$ have their expected types lifted to expressions. For example, *put* :: *Exp s* → *Exp* (*State s* ()) and *get* :: *Exp* (*State s s*).

Normalizing *prgSt* yields the following result.

```
∗NbE.OpenNbE> norm prgSt
λarr.(Get ≫= λs.(Put (NewArr (LenArr arr) (λi.(arr ! i) + 1))
  ≫ return ((arr ! 0) + 1)))
```

The resulting program puts a new array that contains the elements of the original array incremented by 1, and returns the head of the original array, also incremented by 1. The first *put* operation in *prgSt* is removed as it is overwritten by the subsequent *put*. Similarly, the operation *get* and the intermediate array *arr′* in *prgSt* are also removed, as the array in the state is known locally from the previous *put* operation. The *Get* in the result is redundant as the state *s* is never used. This *Get* is introduced by the normalizer as a consequence of η-expansion (see Section 5). We show later, in Section 6, how such redundancy in generated code can be eliminated by disabling η-expansion.

Normalizing branching stateful programs. The presence of side-effects and branching in the same language creates subtle interactions between the primitives that must be considered when implementing normalization. To illustrate that our NbE procedure can also be applied seamlessly to their combination, we consider the following program that combines the last two examples.

```
prgBrSt :: Exp (Either Int Int → Arr Int → State (Arr Int) Int)
prgBrSt = lam $ λscr → lam $ λarr →
  put (mapArr (lam (+1)) arr)
```

```
  -- Expressions, neutrals and normal forms
data Exp a where ...
data Ne a where ...
data Nf a where ...

  -- Embedding functions
embNe :: Ne a → Exp a
embNf :: Nf a → Exp a

  -- NbE semantics
class Rf a where
  type Sem a :: ∗
  reify   :: Sem a → Nf a
  reflect :: Ne a → Sem a

  -- Evaluation function
eval  :: Rf a ⇒ Exp a → Sem a

  -- Normalization function
norm :: Rf a ⇒ Exp a → Exp a
norm = embNf ∘ reify ∘ eval
```

Fig. 1. Components of NbE

```
≫st (case′ scr
  (lam $ λx → put (mapArr (lam (+x)) arr))
  (lam $ λy → return unit))
≫st get ≫=st (lam (λarr′ → returnst (ixArr arr′ 0))))
```

Unlike in *prgSt*, the first *put* here cannot be eliminated as the second branch does not have a subsequent *put*. Moreover, elimination of *get* here is less straightforward as we cannot readily determine the value of the array in the state.

Normalizing *prgBrSt* yields the following result.

```
∗NbE.OpenNbE> norm prgBrSt
λscr.(λarr.(Get ≫= (λs.(Case scr of
  (λx.(Put (NewArr (LenArr arr) (λi.(arr ! i) + x))
    ≫ Return ((arr ! 0) + x)))
  (λy.(Put (NewArr (LenArr arr) (λi.(arr ! i) + 1))
    ≫ Return ((arr ! 0) + 1)))))))
```

The resulting program pattern matches on *scr*, performs appropriate *put* operations and returns the expected result individually on each branch. The first *put* operation, i,e., *put* (*mapArr* (*lam* (+1)) *arr*) is discarded in the first branch but preserved in the latter!

3 NBE FOR AN EDSL CORE

NbE is the process of *evaluating*, or interpreting, expressions of a language in a semantic domain and then obtaining normal forms by *reifying*, or extracting, normal forms from values in the semantic domain. The key idea behind NbE is to leverage

```
data Exp a where
  Var  :: Rf a ⇒ String → Exp a
  Lift :: Base a ⇒ a → Exp a
  Lam  :: (Rf a, Rf b) ⇒ (Exp a → Exp b) → Exp (a → b)
  App  :: (Rf a, Rf b) ⇒ Exp (a → b) → Exp a → Exp b
  Unit :: Exp ()
  Pair :: (Rf a, Rf b) ⇒ Exp a → Exp b → Exp (a, b)
  Fst  :: (Rf a, Rf b) ⇒ Exp (a, b) → Exp a
  Snd  :: (Rf a, Rf b) ⇒ Exp (a, b) → Exp b
  Mul  :: Exp Int → Exp Int → Exp Int
  Add  :: Exp Int → Exp Int → Exp Int
  Rec  :: Rf a ⇒ Exp Int
       → Exp (Int → a → a) → Exp a → Exp a
```

Fig. 2. Basic core expression language

an (often non-standard) evaluator implemented in the host language to normalize expressions in the object language—hence the name normalization *by* evaluation.

Figure 1 summarizes the components of NbE in our implementation. The object language is defined by the expression data type *Exp*, and its normal forms are defined by *Nf* and *Ne* (a subcategory of normal forms called *neutrals*). Unlike a traditional evaluator, an NbE evaluator interprets expressions in a semantic domain that is carefully chosen such that normal forms can be reified from it. The type class *Rf* specifies the requirements of such a semantic domain.

In the class *Rf*, the type family *Sem* maps types in the object language to the Haskell types that interpret them. The definition of *Rf* requires that an interpretation of a type be chosen such that we can also implement the functions *reify* and *reflect*. The function *reify* performs reification, and the function *reflect* performs a process known as *reflection*. Reflection inserts neutral expressions into the semantic domain, and is used to evaluate free variables whose values are unknown. Reflection is crucial to reifying functions: to convert a semantic function to a syntactic one, we apply it to a semantic value given by the reflecting the argument variable of the syntactic function. Our syntax for functions calls for a slightly different treatment, as we shall see shortly.

In this section, we discuss the implementation of NbE for an eDSL core language that is defined by the *Exp* data type. This language is based on a simply-typed lambda calculus (STLC) with product and base types, extended with primitive recursion and simple arithmetic operations. We later extend it further with array and sum types (Section 4), exception and state effects (Section 5), and other uninterpreted primitives (Section 6). These features have been chosen to illustrate the practical applicability, extensibility, and customizability of NbE to a class of functional eDSLs like Feldspar [6], Haski [42], and others [4, 40] found in eDSL literature.

Figure 2 summarizes the pure fragment of the core expression syntax. It consists of expression constructors for unknowns (*Var*), constants (*Lift*), functions (*Lam*, *App*), products (*Pair*, *Fst*, *Snd*), arithmetic operations (*Mul*, *Add*), and primitive recursion

(*Rec*). The constructor *Var* allows us to insert unbound free variables, and *Lift* allows us to lift constant values of primitive base types (identified by the type class *Base*) directly to expressions. For example, instances *Base Int* and *Base String* allow us to lift integers and strings to expressions of type *Exp Int* and *Exp String* respectively.

Function and product types. To implement NbE for a fragment of the language under consideration, we begin by specifying the equations of interest, and identifying normal forms of these equations. The equations for the fragment of function and product types are specified as follows.

$$
\begin{aligned}
&f :: \mathit{Exp}\ (a \to b) \approx \mathit{Lam}\ (\mathit{App}\ f)\\
&\mathit{App}\ (\mathit{Lam}\ f)\ e \ \approx f\ e\\
&p :: \mathit{Exp}\ (a, b) \quad \approx \mathit{Pair}\ (\mathit{Fst}\ p)\ (\mathit{Snd}\ p)\\
&\mathit{Fst}\ (\mathit{Pair}\ e_1\ e_2) \approx e_1\\
&\mathit{Snd}\ (\mathit{Pair}\ e_1\ e_2) \approx e_2
\end{aligned}
$$

The type directed equations, or η-laws, specify the structure of the resulting normal forms, and the reduction laws, or β-laws, specify how expressions should be reduced.

To a first approximation, neutrals denote expressions whose reduction is stuck at unknowns, and normal forms denote value expressions. A normal form in NbE only needs to be some canonical element in the equivalence class of expressions identified by the equations, but it is often helpful to think of it as an expression that cannot be reduced further by applying the β laws by orienting them from left to right, and has a canonical shape as dictated by the η law. For this fragment, we define neutral and normal forms as follows, resembling β-short η-long normal forms in STLC.

```
data Ne a where
  NVar :: Rf a ⇒ String → Ne a
  NApp :: (Rf a, Rf b) ⇒ Ne (a → b) → Nf a → Ne b
  NFst :: (Rf a, Rf b) ⇒ Ne (a, b) → Ne a
  NSnd :: (Rf a, Rf b) ⇒ Ne (a, b) → Ne b
data Nf where
  NUp  :: Base a ⇒ Ne a → Nf a
  NUnit :: Nf ()
  NLam :: (Rf a, Rf b) ⇒ (Exp a → Nf b) → Nf (a → b)
  NPair :: (Rf a, Rf b) ⇒ Nf a → Nf b → Nf (a, b)
```

Observe that a normal form of type $\mathit{Nf}\ (a \to b)$ cannot be reduced further by applying the β-law on any of its subexpressions, and it must be constructed by *NLam*. This property of normal forms can as well be observed for products and all other types under consideration in this paper.

The normal form constructor for functions, *NLam*, receives an argument of type $\mathit{Exp}\ a \to \mathit{Nf}\ b$ instead of the more restrictive type $\mathit{Nf}\ a \to \mathit{Nf}\ b$. This is to allow the *syntactic* embedding—i.e, without invoking functions that involve semantics, such as *eval* or *reify*—of normal forms to expressions via *embNf* by mapping *NLam* to *Lam*, which would not be possible with the latter option.

After the identification of suitable normal forms, it remains to define a semantic domain that supports the reification of normal forms and evaluation of terms. The

semantic domain for product and function types are readily available in Haskell, so we simply interpret them by their Haskell counterparts by defining instances of *Rf* as follows.

```
instance Rf () where
  type Sem () = ()
  reify   _ = NUnit
  reflect _ = ()

instance (Rf a, Rf b) ⇒ Rf (a, b) where
  type Sem (a, b) = (Sem a, Sem b)
  reify   p = NPair (reify (fst p)) (reify (snd p))
  reflect n = (reflect (NFst n), reflect (NSnd n))

instance (Rf a, Rf b) ⇒ Rf (a → b) where
  type Sem (a → b) = Sem a → Sem b
  reify f   = NLam (reify ∘ f ∘ eval)
  reflect n = λy → reflect (NApp n (reify y))
```

The implementation of functions *reify* and *reflect* is achieved by converting from and to Haskell values. To reify a pair $p :: (Sem\ a, Sem\ b)$, we construct a normal form using the constructor *NPair*, whose arguments are obtained by recursively reifying the projections of p. To reflect a neutral $n :: Ne\ (a, b)$, we construct a pair whose components are obtained by recursively reflecting the projections of n using neutral constructors *NFst* and *NSnd*. On the other hand, to reify a function $f :: Sem\ a \to Sem\ b$, we evaluate the expression argument[1] provided by the constructor *NLam* and recursively reify its application to f, and to reflect a neutral $n :: Ne\ (a \to b)$, we recursively reflect the application of n using the constructor *NApp* with the reification of the semantic argument y.

Evaluation resembles a standard evaluator, with the exception of the *Var* and *Lam* cases, as witnessed below.

```
eval (Var x :: Exp a) = reflect@a (NVar x)
eval Unit             = ()
eval (Lam f)          = λy → eval (f (embNf (reify y)))
eval (App f e)        = (eval f) (eval e)
eval (Pair e e')      = (eval e, eval e')
eval (Fst e)          = fst (eval e)
eval (Snd e)          = snd (eval e)
```

For the *Var* case, we use reflection to insert the neutral $NVar\ x$ into the semantics, and for the *Lam* case, we recursively evaluate the application of f to an expression obtained by reifying and embedding the semantic argument y.

Reflection constructs a semantic value based on the type of a neutral, which when reified, has the effect of η-expansion [9]. Evaluating an unknown Var `"x"` $:: Exp\ (() \to ())$ returns its reflection $\lambda y \to ()$, which when reified yields the normal form $NLam\ (\lambda e \to NUnit)$, where η-expansion has been applied for both the function and unit types.

[1] Traditionally, reflection is sufficient since the argument in *Lam* is a variable, but our formulation demands evaluation since it can be any expression.

```
eval (Var "x" :: Exp (() → ()))
  -- by definition
  ≡ reflect@(() → ()) (NVar "x")
  -- reflecting neutral of type ‘Ne (() -> ())‘
  ≡ λy → reflect@() (NApp (NVar x) (reify y))
  -- reflecting neutral of type ‘Ne ()‘
  ≡ λy → ()
reify@(() → ()) (λy → ())
  -- reifying value of type ‘() -> ()‘
  ≡ NLam (reify@() ∘ f ∘ eval)
  -- function composition
  ≡ NLam (λe → reify@() (f (eval e)))
  -- reifying value of type ‘()‘
  ≡ NLam (λe → NUnit)
```

Base types. The expression syntax can be freely extended with base types by defining new instances of the type class *Base*. Normal forms of base types can either be neutrals or values. While neutrals can be embedded into normal forms using the constructor *NUp*, we extend the definition of normal forms with a constructor *NLift* to embed values.

```
data Nf where ...
  NLift :: Base a ⇒ a → Nf a
```

The semantic domain for base types resemble the definition of normal forms as neutrals or values, which we illustrate for the types *Int* and *String* below.

```
instance Rf Int where
  type Sem Int = Either (Ne Int) Int
  reify x      = either NUp NLift x
  reflect n    = Left n

instance Rf String where
  type Sem Int = Either (Ne String) String
    -- similar to above
```

For integers, we use the type *Either* (*Ne Int*) *Int* as the semantic domain for interpretation, and similarly for strings we use *Either* (*Ne String*) *String*. Reification replaces *Left* by *NUp* and *Right* by *NLift*, while reflection embeds a neutral into the semantic domain using *Left*.

In the absence of primitives that return a value of base type, such as *String*, we need not perform any further modifications. For base types with primitives, such as *Int*, however, we must also extend evaluation and the definition of neutrals to accommodate them.

For a simple treatment of integer expressions, let us suppose that we would like to normalize them using the following equations.

```
Add (Lift x) (Lift y) ≈ Lift (x + y)
Mul (Lift x) (Lift y) ≈ Lift (x * y)
```

```
data Exp a where ...
  NewArr :: Rf a ⇒ Exp Int → Exp (Int → a) → Exp (Arr a)
  LenArr :: Rf a ⇒ Exp (Arr a) → Exp Int
  IxArr  :: Rf a ⇒ Exp (Arr a) → Exp Int → Exp a
  Inl    :: (Rf a, Rf b) ⇒ Exp a → Exp (Either a b)
  Inr    :: (Rf a, Rf b) ⇒ Exp b → Exp (Either a b)
  Case   :: (Rf a, Rf b, Rf c) ⇒ Exp (Either a b)
         → Exp (a → c) → Exp (b → c) → Exp c
```

Fig. 3. Extension with arrays and sums

These equations specify that addition and multiplication must be performed when both the operands are available as lifted integer values. In the absence of either, such as in *Add* (*Lift* 2) (*Var* "x"), the expression cannot be reduced further, and must be considered to be in normal form.

To implement these equations, we extend the definition of neutrals for stuck applications of *Add* and *Mul* as follows.

```
data Ne a where ...
  NAdd1 :: Ne Int → Int    → Ne Int
  NAdd2 :: Int    → Ne Int → Ne Int
  NAdd  :: Ne Int → Ne Int → Ne Int
    -- similarly NMul1, NMul2 and NMul
```

Following this, evaluation can be implemented using semantic functions add', mul' :: *Sem Int* → *Sem Int* → *Sem Int* as below. These functions are implemented by performing the corresponding operation when both the right injections are available, and constructing neutrals otherwise.

```
eval (Add e e') = add' (eval e) (eval e')
eval (Mul e e') = mul' (eval e) (eval e')
```

4 NBE FOR ARRAYS AND SUMS

Figure 3 summarizes the extension of the core language with array and sum types. The type *Exp* (*Arr a*) denotes an *a* array expression indexed by integers, and the type *Exp* (*Either a b*) denotes a sum expression of type *Either a b*. The array operation *NewArr* constructs a new array, *LenArr* computes the length of an array, and *IxArr* indexes into an array. Sum types are formulated in the usual way with injections (*Inl* and *Inr*) and case analysis (*Case*).

4.1 Arrays

Array primitives satisfy the following equations, where the first is η-expansion for arrays, and the latter two are reductions for *LenArr* and *IxArr* respectively.

```
arr :: Exp (Arr a)      ≈ NewArr (LenArr arr) (Lam (IxArr arr))
LenArr (NewArr n f) ≈ n
IxArr (NewArr n f) k ≈ App f k
```

Neutral and normal forms are defined by placing stuck applications of *LenArr* and *IxArr* in neutrals, and an array construction using *NewArr* in normal forms.

```
data Ne a where ...
  NLenArr :: Rf a ⇒ Ne (Arr a) → Ne Int
  NIxArr  :: Rf a ⇒ Ne (Arr a) → Nf Int → Ne a
data Nf a where ...
  NNewArr :: Rf a ⇒ Nf Int → (Exp Int → Nf a) → Nf (Arr a)
```

The semantic domain for arrays, defined by *SArr* below, is given by a refinement of a shallow embedding of arrays in Haskell (called *vectors* in Feldspar [6]).

```
data SArr a where
  SNewArr :: Sem Int → (Exp Int → a) → SArr a
instance (Rf a) ⇒ Rf (Arr a) where
  type Sem (Arr a) = SArr (Sem a)
  reify (SNewArr k f) = NNewArr (reify k) (reify ∘ f)
  reflect n           = SNewArr
    (reflect (NLenArr n))
    (reflect ∘ NIxArr n ∘ reify ∘ eval)
```

The constructor *SNewArr* constructs a semantic array from the length of an array, given by a semantic integer *Sem Int*, and a function *Exp Int* → *a* that returns elements of the array for a given index expression. Reification converts a semantic array constructed using *SNewArr* to a syntactic one in normal form constructed using *NNewArr*. Reflection, on the other hand, inserts a neutral $n :: Exp\ (Arr\ a)$ into semantics by constructing a semantic array with the same length and same elements as *n*.

Evaluation is extended to arrays by interpreting *NewArr* as *SNewArr*, and the array operations *IxArr* and *LenArr* by extracting the appropriate components of *SNewArr*.

```
eval (NewArr n f) = SNewArr (eval n) f
eval (IxArr arr i)  = let (SNewArr _ f) = eval arr in f i
eval (LenArr arr)   = let (SNewArr n _) = eval arr in n
```

4.2 Sum Types

Equations and normal forms. Expressions of sum types are given the following standard equations.

```
e :: Exp (Either a b) ≈ Case e Inl Inr
Case (Inl e) f g      ≈ App f e
Case (Inr e) f g      ≈ App g e
F (Case e g h)        ≈ Case e (F ∘ g) (F ∘ h)
```

The first equation specifies a restricted η-expansion for sums. The second and third equations are the standard β-rules for sums. The last equation is a *commuting conversion*, where the function *F* denotes an elimination context, which arises from a more general η-rule [29] and enables more opportunities to apply the β-rules [35]. This equation is further explained in Appendix A.1. Normal forms for sums comprise injections *and case analysis*.

```
data Nf a where ...
  NInl  :: (Rf a, Rf b)
    ⇒ Nf a → Nf (Either a b)
  NInr  :: (Rf a, Rf b)
    ⇒ Nf b → Nf (Either a b)
  NCase :: (Rf a, Rf b, Rf c) ⇒ Ne (Either a b)
    → (Exp a → Nf c) → (Exp b → Nf c) → Nf c
```

Unlike stuck applications of eliminators, such as *NFst* and *NSnd*, that we class as neutral, we classify a stuck application of *NCase* as a normal form. This choice has to do with the implementation of the commuting conversions for sums.

Classifying *NCase* as neutral does not force commuting reductions, and may cause case analysis to prevent reductions by harboring introduction forms. For example, defining *NCase* under neutrals would deem the following expression to be neutral, and thus normal (via *NUp*).

```
NApp (NCase (NVar "x") (NLam id) (NLam id)) (NLift 1)
```

Placing *NCase* in normal forms, on the other hand, forces this expression to be reduced further as below since a normal form of function type cannot be applied.

```
NCase (Var "x") (NLam $ λ_ → Lift 1) (NLam $ λ_ → Lift 1)
```

Semantic domain for sums. It is tempting to interpret sum types by their Haskell counterpart, i.e., *Sem* (*Either a b*) = *Either* (*Sem a*) (*Sem b*). But this interpretation is insufficient for NbE, and does not support reflection. For example, what should be the reflection of the unknown *Var* "x" :: *Exp* (*Either* () ())? We cannot make a choice over the *Left* or *Right* injection! To solve this dilemma, we define a semantic domain that captures branching over neutrals (which subsume unknowns), and use that to interpret sums.

```
data MDec a where
  Leaf   :: a → MDec a
  Branch :: (Rf a, Rf b) ⇒ Ne (Either a b)
    → (Exp a → MDec c) → (Exp b → MDec c) → MDec c

instance Monad MDec where ...

instance (Rf a, Rf b) ⇒ Rf (Either a b) where
  type Sem (Either a b) = MDec (Either (Sem a) (Sem b))
  reify (Leaf (Left x))  = NInl (reify x)
  reify (Leaf (Right x)) = NInr (reify x)
  reify (Branch n f g)   = NCase n (reify ∘ f) (reify ∘ g)
  reflect n = Branch n
    (Leaf ∘ Left ∘ eval)
    (Leaf ∘ Right ∘ eval)
```

Intuitively, the data type *MDec* defines a decision tree (monad) that prevents us from having to make a choice during reflection. Unlike a value of type *Either* (*Sem a*) (*Sem b*), a value of type *MDec* (*Either* (*Sem a*) (*Sem b*)) can be constructed using the *Branch* constructor without making a choice. The *Branch* constructor requires us to handle

both possible injections, and is the semantic equivalent of the normal form *NCase*—as witnessed by the implementation of *reify*.

Evaluating case analysis. The introduction of sum types causes a subtle problem for evaluation: consider the following expression of type *Exp Int*.

```
Case (Var "x") (Lam $ λ_ → Lift 1) (Lam $ λ_ → Lift 2)
```

While irreducible (and representable as a normal form), the semantic domain for integers, i.e., *Either* (*Ne Int*) *Int*, has no room for its interpretation! How should this *Case* expression of type *Expr Int* be evaluated as a value of type *Either* (*Ne Int*) *Int*? We proceed to adapt our interpretation of *Int* (and similarly with *String*) as follows.

```
instance Rf Int where
  type Sem Int = MDec (Either (Ne Int) Int) ...
instance Rf String where
  type Sem Int = MDec (Either (Ne String) String) ...
```

In short, we place the decision tree monad *MDec* on top of the original interpretation of *Int* allowing room for constructing case trees in the semantics. The problematic integer expression from above can now be evaluated to:

```
Branch (NVar "x") (λ_ → Right 1) (λ_ → Right 2)
```

Reification and reflection can be implemented easily by adapting our previous implementation to deal with *MDec* along the lines of the *Rf* (*Either a b*) instance.

Following this change to the interpretation, we proceed with evaluation as below using a semantic function *run* :: *Rf a* ⇒ *MDec* (*Sem a*) → *Sem a* that can be implemented by induction on the type parameter *a*—which rephrases the branches of the decision tree as semantic ones.

```
eval (Inl e)       = Leaf (Left (eval e))
eval (Inr e)       = Leaf (Right (eval e))
eval (Case s f g) = let s' = eval s; f' = eval f; g' = eval g
  in run (fmap (either f' g') s')
```

Not all type constructors require a modification of the semantic domain. In particular, all type constructors with a single introduction form and a corresponding η-rule (such as functions, products, and arrays) avoid this problem as we may perform η-expansion of the *Case* expression, followed by commuting conversions, to represent the value in the semantic domain. For example, the following expression of type *Exp* (*Int*, *Int*)

```
Case (Var "x") (Lam $ λ_ → Var "y") (Lam $ λ_ → Var "z")
```

can be η-expanded and then two commuting conversions applied to give

```
Pair
  (Case (Var "x")
     (Lam $ λ_ → Fst (Var "y")) (Lam $ λ_ → Fst (Var "z")),
   Case (Var "x")
     (Lam $ λ_ → Snd (Var "y")) (Lam $ λ_ → Snd (Var "z")))
```

which is interpreted as a pair of semantic integers:

```
data Exp a where ...
  Throw     :: Rf a ⇒ Exp String → Exp (Err a)
  Catch     :: Rf a ⇒ Exp (Err a)
            → Exp (String → Err a) → Exp (Err a)
  Return_err :: Rf a ⇒ Exp a → Exp (Err a)
  Bind_err   :: (Rf a, Rf b) ⇒ Exp (Err a)
            → Exp (a → Err b) → Exp (Err b)
```

(a) Exceptions

```
data Exp a where ...
  Get       :: Rf s ⇒ Exp (State s s)
  Put       :: Rf s ⇒ Exp s → Exp (State s ())
  Return_st :: (Rf s, Rf a) ⇒ Exp a → Exp (State s a)
  Bind_st   :: (Rf s, Rf a, Rf b) ⇒ Exp (State s a)
            → Exp (a → State s b) → Exp (State s b)
```

(b) State

Fig. 4. Extension with exception and state effects

```
(Branch (NVar "x")
    (λ_ → NFst (NVar "y")) (λ_ → NFst (NVar "z")),
 Branch (NVar "x")
    (λ_ → NSnd (NVar "y")) (λ_ → NSnd (NVar "z")))
```

This means that we need only to refine our interpretation of *Int* and *String*, where we lack a combination of a single introduction form accompanied by a corresponding η-rule.

The above treatment of sums is sound and often suffices in practice, but it does not capture all natural equations for sums. In Section 6 we outline how to augment our implementation to eliminate repeated and redundant case splits.

5 NBE FOR MONADIC EFFECTS

Figure 4 summarizes the extension of the expression syntax respectively with exceptions and state formulated as monadic types. Exceptions consist of a throw operation (*Throw*) to throw string exceptions, a catch operation (*Catch*) to handle exceptions, along with the return ($Return_{err}$), and bind ($Bind_{err}$) of the monadic type *Err*. Stateful computations are formulated similar to the State monad in Haskell, and consists of a get operation (*Get*) to retrieve the state, a put operation (*Put*) to overwrite the state, along with the return ($Return_{st}$), and bind ($Bind_{st}$) of the monadic type *State s*.

Both monadic types (denoted M) are subject to the following equations, typically called the *monad laws*.

$$
\begin{aligned}
&m :: Exp\ (M\ a) && \approx Bind\ m\ Return\\
&Bind\ (Return\ x)\ f && \approx App\ f\ x\\
&Bind\ (Bind\ e_1\ f)\ g && \approx Bind\ e_1\ (Lam\ (\lambda x \to Bind\ (App\ f\ x)\ g))
\end{aligned}
$$

The first equation is η-expansion for monads, the second β-reduction, and the third a commuting conversion that arranges *Bind* operations in a right-associative chain.

5.1 Exceptions

As well as the monad laws, exception computations also obey the following equations. The first equation is η-expansion and the second and third equations are β-reductions for exceptions.

$$
\begin{aligned}
&m :: \mathit{Exp}\ (\mathit{Err}\ a) &&\approx \mathit{Catch}\ m\ \mathit{Throw}\\
&\mathit{Catch}\ (\mathit{Throw}\ s)\ f &&\approx \mathit{App}\ f\ s\\
&\mathit{Catch}\ (\mathit{Return}\ x)\ f &&\approx \mathit{Return}\ x
\end{aligned}
$$

Notice here that there is a contention between two η laws: one for the *Err* monad and one specific to exceptions. What should be the η-expanded form of an expression $e :: \mathit{Exp}\ (\mathit{Err}\ a)$? We must make a choice here, and we choose $\mathit{Catch}\ (\mathit{Bind}_{\mathit{err}}\ e\ \mathit{Return})\ \mathit{Throw}$, where we first apply the η-rule for monads, and then apply the one for exceptions. Our normal forms reflect this choice using a normal form constructor *NTryUnless* that denotes a fusion of $\mathit{Bind}_{\mathit{err}}$ and *Catch* in normal form [8].

$$
\begin{aligned}
&\textbf{data}\ \mathit{Nf}\ a\ \textbf{where}\ \ldots\\
&\quad \mathit{NReturn}_{\mathit{err}}\ :: \mathit{Rf}\ a \Rightarrow \mathit{Nf}\ a \to \mathit{Nf}\ (\mathit{Err}\ a)\\
&\quad \mathit{NThrow}\ :: \mathit{Rf}\ a \Rightarrow \mathit{Nf}\ \mathit{String} \to \mathit{Nf}\ (\mathit{Err}\ a)\\
&\quad \mathit{NTryUnless} :: (\mathit{Rf}\ a, \mathit{Rf}\ b) \Rightarrow \mathit{Ne}\ (\mathit{Err}\ a)\\
&\qquad \to (\mathit{Exp}\ a \to \mathit{Nf}\ (\mathit{Err}\ b))\\
&\qquad \to (\mathit{Exp}\ \mathit{String} \to \mathit{Nf}\ (\mathit{Err}\ b)) \to \mathit{Nf}\ (\mathit{Err}\ b)
\end{aligned}
$$

The constructors $\mathit{NReturn}_{\mathit{err}}$ and *NThrow* are the normal form counterparts of $\mathit{Return}_{\mathit{err}}$ and *Throw*.

The semantic domain is defined by a data type *MErr* that closely parallels the structure of normal forms.

$$
\begin{aligned}
&\textbf{data}\ \mathit{MErr}\ a\ \textbf{where}\\
&\quad \mathit{SReturn}_{\mathit{err}}\ :: \mathit{Rf}\ a \Rightarrow a \to \mathit{MErr}\ a\\
&\quad \mathit{SThrow}\ :: \mathit{Rf}\ a \Rightarrow \mathit{Nf}\ \mathit{String} \to \mathit{MErr}\ a\\
&\quad \mathit{STryUnless} :: (\mathit{Rf}\ a, \mathit{Rf}\ b) \Rightarrow \mathit{Ne}\ (\mathit{Err}\ a)\\
&\qquad \to (\mathit{Exp}\ a \to \mathit{MErr}\ b)\\
&\qquad \to (\mathit{Exp}\ \mathit{String} \to \mathit{MErr}\ b) \to \mathit{MErr}\ b\\
&\textbf{instance}\ (\mathit{Rf}\ a) \Rightarrow \mathit{Rf}\ (\mathit{Err}\ a)\ \textbf{where}\\
&\quad \textbf{type}\ \mathit{Sem}\ (\mathit{Err}\ a) = \mathit{MErr}\ (\mathit{Sem}\ a)\\
&\quad \mathit{reify}\ (\mathit{SReturn}_{\mathit{err}}\ x) = \mathit{NReturn}_{\mathit{err}}\ (\mathit{reify}\ x)\\
&\quad \mathit{reify}\ (\mathit{SThrow}\ n) = \mathit{NThrow}\ n\\
&\quad \mathit{reify}\ (\mathit{STryUnless}\ n\ f\ g) = \mathit{NTryUnless}\ n\ (\mathit{reify} \circ f)\ (\mathit{reify} \circ g)\\
&\quad \mathit{reflect}\ n = \mathit{STryUnless}\ n\ (\mathit{SReturn}_{\mathit{err}} \circ \mathit{eval})\ (\mathit{SThrow} \circ \mathit{eval})
\end{aligned}
$$

The data type definition of *MErr* gives rise to a semantic monad that can be used to evaluate the monadic expression constructors $\mathit{Return}_{\mathit{err}}$ and $\mathit{Bind}_{\mathit{err}}$. We evaluate *Throw* using semantic constructor *SThrow*, and *Catch* using a semantic function *catch*′ that is implemented by pattern matching on its first argument.

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Return}_{\mathit{err}}\ e) = \mathit{return}\ (\mathit{eval}\ e)\\
&\mathit{eval}\ (\mathit{Bind}_{\mathit{err}}\ e\ f) = \mathit{eval}\ e \gg\!= \mathit{eval}\ f\\
&\mathit{eval}\ (\mathit{Throw}\ e) \quad = \mathit{SThrow}\ (\mathit{eval}\ e)\\
&\mathit{eval}\ (\mathit{Catch}\ e\ f) \quad = \mathit{catch}'\ (\mathit{eval}\ e)\ (\mathit{eval}\ f)
\end{aligned}
$$

instance *Monad MErr* **where** ...
$\mathit{catch}' :: \mathit{MErr}\ \mathit{sa} \to (\mathit{Sem}\ \mathit{String} \to \mathit{MErr}\ \mathit{sa}) \to \mathit{MErr}\ \mathit{sa}$

The rest of the definitions can be found in Appendix A.3.

5.2 Stateful Computations

Similar to exceptions, stateful computations are also given equations specific to the operations *Put* and *Get*, in addition to the monad laws.

$$
\begin{aligned}
&m :: \mathit{Exp}\ (\mathit{State}\ s\ a) \approx \mathit{Get} \gg\!=_{\mathit{st}} (\mathit{Lam}\ (\lambda s \to (\mathit{Put}\ s) \gg_{\mathit{st}} m))\\
&(\mathit{Put}\ x) \gg_{\mathit{st}} ((\mathit{Put}\ y) \gg_{\mathit{st}} m) \approx (\mathit{Put}\ y) \gg_{\mathit{st}} m\\
&(\mathit{Put}\ x) \gg_{\mathit{st}} (\mathit{Bind}_{\mathit{st}}\ \mathit{Get}\ f) \quad \approx (\mathit{Put}\ x) \gg_{\mathit{st}} (\mathit{App}\ f\ x)
\end{aligned}
$$

We have an η law as usual, and two reduction laws that reduce sequencing of *Put* and *Get* operations. Note that the operator $\gg\!=_{\mathit{st}}$ is an alias for $\mathit{Bind}_{\mathit{st}}$, and $\gg_{\mathit{st}}$ is a shorthand defined as $m \gg_{\mathit{st}} m' = \mathit{Bind}_{\mathit{st}}\ m\ (\mathit{Lam}\ (\lambda\_ \to m'))$.

As with exceptions, there is a contention between two η-laws, and we choose the η-expanded form of an expression $m :: \mathit{Exp}\ (\mathit{State}\ s\ a)$ to be

$$
\begin{aligned}
&\mathit{Get} \gg\!=_{\mathit{st}} (\mathit{Lam}\,\$\,\lambda s \to (\mathit{Put}\ s) \gg_{\mathit{st}} (m \gg\!=_{\mathit{st}} (\mathit{Lam}\,\$\,\lambda x \to\\
&\quad \mathit{Get} \gg\!=_{\mathit{st}} (\mathit{Lam}\,\$\,\lambda s' \to (\mathit{Put}\ s') \gg_{\mathit{st}} (\mathit{Return}_{\mathit{st}}\ x)))))
\end{aligned}
$$

Our normal forms reflect this choice, while also ensuring that the expression they denote cannot be further reduced by the β-laws.

data *Nf a* **where** ...
$\quad \mathit{NGetPut} :: (\mathit{Rf}\ s, \mathit{Rf}\ a)$
$\qquad \Rightarrow (\mathit{Exp}\ s \to (\mathit{Nf}\ s, \mathit{NfSt}_{\mathit{res}}\ s\ a)) \to \mathit{Nf}\ (\mathit{State}\ s\ a)$

data $\mathit{NfSt}_{\mathit{res}}\ s\ a$ **where**
$\quad \mathit{NReturn}_{\mathit{st}} :: (\mathit{Rf}\ s, \mathit{Rf}\ a) \Rightarrow \mathit{Nf}\ a \to \mathit{NfSt}_{\mathit{res}}\ s\ (\mathit{State}\ s\ a)$
$\quad \mathit{NBind}_{\mathit{st}} \quad :: (\mathit{Rf}\ s, \mathit{Rf}\ a, \mathit{Rf}\ b) \Rightarrow \mathit{Ne}\ (\mathit{State}\ s\ a)$
$\qquad \to (\mathit{Exp}\ a \to \mathit{Nf}\ (\mathit{State}\ s\ b)) \to \mathit{NfSt}_{\mathit{res}}\ s\ (\mathit{State}\ s\ b)$

The data type $\mathit{NfSt}_{\mathit{res}}$ defines a separate syntactic category of normal forms to capture the following desired shape.

$$
\begin{aligned}
&\mathit{NGetPut}\,\$\,\lambda s_1 \to (s_1', \mathit{NBind}_{\mathit{st}}\ n_1\ (\mathit{Lam}\,\$\,\lambda e_1 \to\\
&\quad \mathit{NGetPut}\,\$\,\lambda s_2 \to (s_2', \mathit{NBind}_{\mathit{st}}\ n_2\ (\mathit{Lam}\,\$\,\lambda e_2 \to\\
&\qquad ...\\
&\qquad\quad \mathit{NReturn}_{\mathit{st}}\ x\ \ ...))))
\end{aligned}
$$

Intuitively, a normal form of a state computation is a function constructed using *NGetPut* that gets the global state and returns a new state to put along with a chain of neutrals bound using $\mathit{NBind}_{\mathit{st}}$ ending with $\mathit{NReturn}_{\mathit{st}}$. The constructor $\mathit{NBind}_{\mathit{st}}$ denotes a stuck binding, and $\mathit{NReturn}_{\mathit{st}}$ returns a value in the monad. Since the binding of a neutral may change the state, the definition of normal forms must allow the state to be retrieved and modified after every binding.

The semantic domain is given by data types *MSt* and MSt_{res} that once again parallel the structure of normal forms.

```
newtype MSt s a = SGetPut {
  runMSt :: Sem s → (Sem s, MSt_res s a) }

data MSt_res s a where
  SReturn_st :: (Rf s, Rf a) ⇒ a → MSt_res s a
  SBind_st   :: (Rf s, Rf a, Rf b) ⇒ Ne (State s a)
             → (Exp a → MSt s b) → MSt_res s b

instance (Rf s, Rf a) ⇒ Rf (State s a) where
  type Sem (State s a) = MSt s (Sem a)
  reify m  = NGetPut $ (λ(s, r) → (reify s, reify_res r))
           ∘ runMSt m ∘ eval
    where
      reify_res :: MSt_res s (Sem a) → NfSt_res s a
      reify_res (SReturn_st x) = NReturn_st (reify x)
      reify_res (SBind_st n f) = NBind_st n (reify ∘ f)
  reflect n = SGetPut $ λs → (s, SBind_st n $ λe →
      SGetPut $ λs′ → (s′, SReturn_st (eval e)))
```

The interpretation of *State s a* as *MSt s* (*Sem a*), along with the definition of *MSt* and MSt_{res} lends itself naturally to both reification and reflection.

Evaluation makes use of a monad instance for *MSt s* (defined in Appendix A.3) for $Return_{st}$ and $Bind_{st}$, and the *Get* and *Put* constructs are evaluated using a combination of the semantic constructors *SGetPut* and $SReturn_{st}$.

```
eval (Return_st e) = return (eval e)
eval (Bind_st e e′) = eval e >>= eval e′
eval (Get e)       = SGetPut $ λs → (s, SReturn_st s)
eval (Put e)       = SGetPut $ λ_ → (eval e, SReturn_st ())

instance Monad (MSt s) where ...
```

5.3 Interaction with Sum Types

As in the pure case, the semantic domains with effects also require refinement to account for sums. Unlike in the pure case, it is insufficient to merely place the monad *MDec* on top of the existing interpretation and requires a careful consideration of the monadic operations. This is because case analysis can also be performed on the result of a monadic bind and in between operations.

For the *MErr* monad, case distinction can be performed on the result of a monadic bind, and we extend the data type definition with a constructor similar to *Branch* to allow this.

```
data MErr a where ...
  SCaseErr :: (Rf a, Rf b) ⇒ Ne (Either a b)
    → (Exp a → MErr c)
    → (Exp b → MErr c) → MErr c
```

The constructor *SCaseErr* is reified using the normal form constructor *NCase*.

For the *MSt* monad, on the other hand, case distinction can be performed both on the result of a monadic bind and on the result of a retrieving the state using *SGetPut*. We modify the definition of *MSt* as follows, by placing the *MDec* monad on the result of the functional argument to *SGetPut*.

```
newtype MSt s a = SGetPut {
  runMState :: Sem s → MDec (Sem s, MSt_res s a) }
```

To retain reification, we modify the definition of normal forms in a similar fashion.

```
data Nf a where ...
  NGetPut :: (Rf s, Rf a)
          ⇒ (Exp s → MDec (Nf s, NfSt_res s a)) → Nf (State s a)
```

The modifications performed in this section do not preclude the implementation of semantic functions such as *return*, (≫=), *catch′*, etc., (see Appendix A.3), or the embedding functions *embNe* and *embNf*.

6 PRACTICAL NBE EXTENSIONS AND VARIATIONS

The normalization procedures described in previous sections are adaptations of NbE for simply typed lambda calculus, that strive to identify the normal form of an expression as a canonical element of its equivalence class of semantically identical expressions. This traditional approach to NbE suffers from the following problems for practical eDSL applications:

- Our implementation β-reduces expressions as much as possible and η-expands expressions, yielding normal forms that are in β-short η-long form. Such aggressive normalization can lead to unnecessary code explosion, which may be harmful for code-generating eDSLs.
- The treatment of base types in Section 3 is insufficient for many practical applications. For example, the expression *Add* (*Var* "x") (*Lift* 0) is not reduced, while we would typically like it to be reduced to (*Var* "x").
- We have not yet explained how to incorporate *uninterpreted primitives*, that is, primitives without equations that dictate their behaviour.

In this section, we show these three problems can be addressed by refining the semantic domain used to implement NbE. Specifically, we present techniques to tame code expansion in NbE, a variation of NbE for integers that performs more advanced arithmetic reductions, and a recipe for adding uninterpreted primitives.

6.1 Taming Code Expansion

Disabling η-expansion using glueing. While η-expansion can be useful for some applications such as deciding program equivalence, it may be unsuitable for other applications such as code generation. For example, observe how the normalizer η expands the unknown *Var* "f" :: *Exp* (*Int* → *Arr Int*).

```
∗NbE.OpenNbE> norm (Var "f" :: Exp (Int → Arr Int))
λx.(NewArr (LenArr (f x)) (λi.(f x ! i)))
```

Our implementation of NbE applies η-expansion by default, but we show here how η-expansion can be selectively disabled using the *glueing* technique [17], yielding (potentially) smaller normal forms.

We begin by modifying our definition of normal forms to allow neutrals to be embedded directly.

```
data Nf a where ...
  NUp :: Ne a → Nf a
```

We remove the type constraint *Base a* on the constructor *NUp*, which relaxes the definition of normal forms to include, for example, the unknown *Var* `"f"` above as *NUp* (*NVar* `"f"`).

Let us suppose that we would like to disable η expansion for function types. We refine the semantic domain of function types to include a syntactic component by "glueing" (i.e. pairing) it with normal forms of the function type as follows.

```
instance (Rf a, Rf b) ⇒ Rf (a → b) where
  type Sem (a → b) = (Sem a → Sem b, Nf (a → b))
  reify   = snd
  reflect n = (..., NUp n)
```

Here we write ellipsis (...) for the original implementation of reflection. Reification projects the second component, a normal form, and reflection is modified to include an embedding of the neutral *n* to normal forms.

We proceed with evaluation as follows.

```
eval (Lam f)   = (..., NLam (reify ∘ eval ∘ f))
eval (App f e) = (fst (eval f)) (eval e)
```

For the case of *Lam*, we retain our previous implementation for the first component, and build a normal form in the second component. The evaluation of application is as before, with a minor modification that projects out the semantic function from the recursive evaluation of the expression f.

We may also disable η-expansion for the other types by modifying the interpretation similarly.

```
type Sem (a, b)   = ((Sem a, Sem b), Nf (a, b))
type Sem (Arr a) = (SArr (Sem a), Nf (Arr a))
...
```

Glueing provides a compositional solution to disabling η-expansion for some (or all) types without changing the implementation for other types. In contrast, another approach described by Lindley [28], where, for instance, the type $a \to b$ is interpreted by *Either* (*Sem a* → *Sem b*) (*Ne* (*a* → *b*)), requires a more involved reimplementation of the evaluator. Glueing can also be applied for effect types, but the definition of normal forms requires more careful consideration to avoid unnecessary expansion. Unlike in a strict language, the implementation of glueing in Haskell avoids a significant performance cost as the semantic and syntactic parts are only computed as required, thanks to lazy evaluation.

Controlling duplication with explicit sharing. Much like other program specialization techniques, NbE can cause code duplication. For example, consider a function *double* that doubles its argument as $Lam\ (\lambda x \to Add\ x\ x)$. Normalizing an application of *double* to an irreducible expression *large* causes it to be duplicated as *Add large large*.

Code duplication can be avoided with a *Let* construct for explicit sharing, for which NbE can be extended as follows.

```
data Exp a where ...
  Let :: (Rf a, Rf b) ⇒ Exp a → Exp (a → b) → Exp b

data Ne a where ...
  NLet :: (Rf a, Rf b) ⇒ Nf a → Nf (a → b) → Ne b

eval (Let e f) = reflect (NLet (reify (eval e)) (reify (eval f)))
```

Normalizing *Let* expressions respects sharing, and the expression *Let large double* does not reduce, avoiding duplication.

Disabling normalization on subexpressions. An alternative to explicit sharing is to disable normalization entirely on a subexpression using a construct *Save*, such that normalizing *Save* (*App double large*) returns the original expression unaffected[2]. We achieve this with a *Save* construct as follows.

```
data Exp a where ...
  Save :: Exp a → Exp a

data Ne a where ...
  NSave :: Exp a → Ne a

eval (Save e) = reflect (NSave e)
```

Optimizing case expressions. The implementation of commuting conversions for sums in Section 4 can produce normal forms with redundant or repeated case analysis.

```
Case scr (Lam $ λ_ → e) (Lam $ λ_ → e)
Case scr (Lam $ λx → Case scr...) (Lam $ λy → Case scr...)
```

The use of *Case* in these expressions is wasteful, and can be optimized further to reduce the size of the generated normal forms. Specifically, we are interested in the following two equations (identified by Lindley [29] as constituents of the general η-rule for sums).

```
Case scr (Lam $ λ_ → e) (Lam $ λ_ → e) ≈ e
Case scr (Lam $ λx → Case scr f1 f2)
         (Lam $ λy → Case scr g1 g2)
          ≈ Case scr (Lam $ λx → f1) (Lam $ λy → g2)
```

The first equation removes a redundant case analysis on *scr*, while the second removes a repeated analysis on *scr*.

One way to implement these equations is to refine the definition of *MDec* to preclude problematic decision trees by construction. However, given Haskell's limited support for dependent types and the pervasive nature of NbE for sums, this is a

[2] in an implementation that disables η expansion entirely

somewhat non-trivial modification. An easier (albeit ad hoc) solution is to implement a post-processing function $optimize :: Rf\ a \Rightarrow MDec\ (Nf\ a) \rightarrow MDec\ (Nf\ a)$ that is invoked when reifying decision trees. This is made possible since these transformations are merely syntactic manipulations of case trees that introduce no further reductions.

6.2 Applying Arithmetic Equations

To implement richer arithmetic equations (specified in Appendix A.1), our selection of normal forms must force the normalizer to perform reductions specified by these equations, for example, by reducing $Add\ (Lift\ 1)\ (Lift\ 2)$ to $Lift\ 3$, $Add\ (Lift\ 0)\ (Var\ \texttt{"x"})$ to $Var\ \texttt{"x"}$, and so on.

We consider normal forms of integers to be in a sum-of-products form $(a_k * n_k) + (a_{k-1} * n_{k-1}) + ... + a_0$, where a_i denotes a constant, and n_i denotes a neutral expression, for each i. Correspondingly, we extend the definition of neutrals and normal forms as follows.

```
data Ne a where ...
  NMul :: Ne Int → Ne Int → Ne Int
data Nf a where ...
  NInt  :: Int → Nf Int
  NAdd :: (Int, Ne Int) → Nf Int → Nf Int
```

The *NMul* constructor in neutrals denotes a stuck multiplication, and *NAdd* denotes the addition of an integer $(a_i * n_i)$ to the left end of an integer in sum-of-products form.

We define the semantic domain for integers using a data type *SOPInt* that is identical to the shape of normal forms, and use it in our definition of an instance of *Rf*.

```
data SOPInt where
  SInt :: Int → SOPInt
  SAdd :: (Int, Ne Int) → SOPInt → SOPInt

instance Rf Int where
  type Sem Int          = SOPInt
  reify (SInt a0)       = NInt a0
  reify (SAdd (ai, ni) k) = NAdd (ai, ni) (reify k)
  reflect n             = SAdd (1, n) (SInt 0)
```

The implementation of *reify* simply converts from the semantic domain to normal forms, while *reflect* expands a neutral $n :: Exp\ Int$ to the form $(1 * n) + 0$.

Evaluation can be implemented by interpreting *Add* and *Mul* by their semantic counterparts add' and mul', which can be defined by induction on values of *SOPInt*.

```
add' :: SOPInt → SOPInt → SOPInt
mul' :: SOPInt → SOPInt → SOPInt

eval (Add e1 e2) = add' (eval e1) (eval e2)
eval (Mul e1 e2) = mul' (eval e1) (eval e2)
```

The function *add′* adds two integers $(a_k * n_k) + ... + a_0$ and $(b_j * m_j) + ... + b_0$ in sum-of-products form by joining them as $(a_k * n_k) + ... + (b_j * m_j) + ... + (a_0 + b_0)$, and function *mul′* multiplies them as $(a_k * b_j) * (n_k * m_j) + (a_k * b_{j-1}) * (n_k * m_{j-1}) + ... + (a_0 * b_0)$.

6.3 Adding Uninterpreted Primitives

The core eDSL can be freely extended with uninterpreted primitives using the unknown constructor *Var*. For example, to extend our eDSL with a fixed-point construct without the corresponding equation, we define a combinator *fix* as:

```
fix :: Rf a ⇒ Exp ((a → a) → a)
fix = Var "Fix"
```

Normalizing an application *fix f* returns the equivalent of the expression of *fix* (*embNf* (*norm f*)), normalizing the function *f*, but leaving *fix* uninterpreted.

7 RELATED WORK

The NbE technique goes back at least as far as Martin-Löf [31] who used it for proving normalization in his work on intuitionistic type theory. The core NbE algorithm for STLC was pioneered by Berger and Schwichtenberg [10]. The name is due to Berger et al. [9] who used it to speed up the Minlog theorem prover. A closely related technique is type-directed partial evaluation (TDPE) [18, 20, 21]. TDPE amounts to an instance of NbE used for partial evaluation in which the NbE semantics is exactly that of the host language. In contrast, for embedding DSLs we make essential use of non-standard semantics, e.g. using glueing for suppressing η-expansion.

Normalization for pure call-by-name STLC with sums is notoriously subtle [23] because general η-rule for sums includes additional equations such as those described in Section 6. Altenkirch et al. [3] give an NbE algorithm for sums based on a *Grothendieck topology* which implicitly captures the kind of decision tree that we use, but at every type. Balat et al. [7], in contrast, make use of multiprompt delimited control to allow retrospective exploration of different branches during reification. Both algorithms build in a degree of syntactic manipulation in order to manage redundant and repeated case splits similarly to what we describe in Section 6.

NbE for sums becomes considerably easier in an effectful call-by-value setting, as fewer equations hold. Danvy [18] uses (single prompt) delimited control operators for handling sums in TDPE. Filinski [22] adapts Danvy's approach to computational lambda calculus extended with sums. Lindley [30] adapts Filinksi's work to replace delimited control with an *accumulation* monad which we here call a *decision tree* monad and Abel and Sattler [1] characterise as a *cover* monad. The Danvy/Filinski approach based on delimited control is at the heart of the treatment of sums in existing eDSLs [40].

Ahman and Staton [2] give an NbE algorithm for general algebraic effects. We speculate that our bespoke treatment of specific monadic effects can be related to their generic approach, but we do not know to what extent their approach maps conveniently onto the Haskell eDSL setting.

Yallop et al. [43] cast partially-static data as free extensions of algebras, which they use as the basis for a generic partial evaluation library, *frex*. The frex approach has

similarities with NbE, providing in particular a principled foundation for optimising in the presence of first-order algebraic theories.

Implementations of NbE in Haskell are not new. For instance, Danvy et al. [19] give an implementation not dissimilar to ours for plain STLC. Prior work on combining deep and shallow embeddings [40] implicitly uses a restricted form of NbE. Their *Syntactic* type class plays a similar role to our *Rf* type class. However, they do not make a connection with NbE and they do not use an instance for functions.

We have presented NbE as a unifying framework for eDSLs based on solid theoretical foundations. A related framework is offered by quoted domain-specific languages (QDSLs) [33]. QDSLs exploit a similar normalization procedure as part of the embedding process. A key difference is that QDSLs are based on staging and a separate normalization algorithm.

The idea of viewing eDSLs through the lens of NbE was explored in an earlier draft paper [32] using Agda rather than Haskell.

8 FINAL REMARKS

We have presented, to the best of our knowledge, the first comprehensive practical implementation of NbE for Haskell eDSLs. NbE provides a systematic and modular approach to specialize eDSL programs in Haskell, and provides a principled account of ad hoc techniques previously developed using a combination of deep and shallow embedding. We have shown how problems that arise from a traditional approach to NbE can be addressed to suit practical concerns such as code expansion, normalization with domain-specific equations, and extension with uninterpreted primitives.

We have not proved the correctness of our NbE implementation, which is typically achieved by showing that an expression is equivalent to its normal form in the chosen equational theory. Moreover, the account of interactions between effects and sums is quite intricate, and appears to be somewhat ad hoc in this level of presentation. A formal investigation of the semantic monads and their interaction is required to identify a more modular solution to add effects to an eDSL that enjoys the benefits of NbE. We leave both these formal aspects as avenues for future work.

We believe that NbE has a broader applicability beyond the examples of fusion shown here. For example, NbE could be used in the security domain, to automatically remove superfluous security checks performed at runtime by programs written in a security eDSL (e.g., [39]). Similarly, in databases (e.g., [37]), NbE could be used normalize queries written in a higher-order eDSL to achieve elimination of higher-order functions and other intermediate data-structures [15, 25].

ACKNOWLEDGMENTS

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023), the Swedish research agency Vetenskapsrådet, and UKRI Future Leaders Fellowship MR/T043830/1 (EHOP).

A APPENDIX

A.1 Equational Theory

Commmuting conversions.

$$F\ (Case\ e\ g\ h) \approx Case\ e\ (F \circ g)\ (F \circ h)$$

This equation specifies commuting conversions that enable us to push eliminators under a case expression, for example, as:

$$App\ f\ (Case\ e\ g\ h) \approx Case\ e\ (App\ f \circ g)\ (App\ f \circ h)$$

The symbol F in the equation is a unary function on expressions that denotes an elimination context such as $App\ f :: Exp\ a \to Exp\ b$ (for some $f :: Exp\ (a \to b)$), $Fst :: Exp\ (a, b) \to Exp\ a$, or $Add\ e :: Exp\ Int \to Exp\ Int$, etc.

Arithmetic equations.

$$\begin{aligned}
(Lift\ x)\ &+ (Lift\ y) &&\approx Lift\ (x + y)\\
(Lift\ 0)\ &+ e &&\approx e\\
(Lift\ x)\ &+ (e_1 + e_2) &&\approx e_1 + (Lift\ x + e_2)\\
(e_1 + e_2)\ &+ e_3 &&\approx e_1 + (e_2 + e_3)\\
(Lift\ x)\ &* (Lift\ y) &&\approx Lift\ (x * y)\\
(Lift\ 0)\ &* e &&\approx Lift\ 0\\
(Lift\ 1)\ &* e &&\approx e\\
(Lift\ x)\ &* (e_1 + e_2) &&\approx (Lift\ x * e_1) + (Lift\ x * e_2)\\
(e_1 + e_2)\ &* e_3 &&\approx (e_1 * e_3) + (e_2 * e_3)
\end{aligned}$$

A.2 Normalizing Primitive Recursion

The recursion construct *Rec* can be used to perform primitive recursion. As mentioned earlier for its combinator counterpart *rec*, an expression $Rec\ n\ f\ x$ is the equivalent of applying f repetitively as $f\ 1\ (f\ 2\ (...(f\ n\ x)))$. This behaviour can be specified by the following equations.

$$\begin{aligned}
&Rec\ i\ f\ x &&\approx x \quad \text{-- (i <= 0)}\\
&Rec\ (e_1 + e_2)\ f\ x &&\approx Rec\ e_1\ f\ (Rec\ e_2\ f\ x)
\end{aligned}$$

To extend our NbE implementation with recursion, we extend the definition of neutrals with a new constructor for stuck recursion as follows.

```
data Ne a where ...
  NRec :: Rf a ⇒ (Int, Ne Int)
    → (Exp Int → Exp a → Nf a) → Nf a → Ne a
```

We then evaluate recursion using a semantic function *rec′*.

```
rec′ :: Rf a ⇒ SOPInt
  → (Sem Int → Sem a → Sem a) → Sem a → Sem a
rec′ (SInt i) f x
  | i ⩽ 0      = x
  | otherwise = rec′ (SInt (i − 1)) f (f (SInt i) x)
rec′ (SAdd aini k) f x
  = reflect (NRec aini f′ (reify (rec′ k f x)))
```

```
    where
      f' i b = reify (f (eval i) (eval b))
eval (Rec n f x) = rec' (eval n) (eval f) (eval x)
```

When the value of an integer is available, *rec′* performs the expected recursion, and otherwise simply applies the second equation of recursion.

A.3 Semantic Monads

```
instance Monad MDec where
  return x = Leaf x
  (Leaf x)         >>= f = f x
  (Branch n g h) >>= f = Branch n ((=<<) f ∘ g) ((=<<) f ∘ h)

instance Monad MErr where
  return x = SReturn_err x
  (SReturn_err x)       >>= f = f x
  (SThrow x)            >>= f = SThrow x
  (STryUnless n g h) >>= f = STryUnless n
    ((=<<) f ∘ g) ((=<<) f ∘ h)
  (SCaseErr n g h)   >>= f = SCaseErr n
    ((=<<) f ∘ g) ((=<<) f ∘ h)

catch' :: MErr sa → (Sem String → MErr sa) → MErr sa
catch' (SReturn_err x)    f = SReturn_err x
catch' (SThrow x)         f = f x
catch' (STryUnless n g h) f = STryUnless n
  (flip catch' f ∘ g) (flip catch' f ∘ h)
catch' (SCaseErr n g h) f = SCaseErr n
  (flip catch' f ∘ g) (flip catch' f ∘ h)

  -- mutually recursive Functor instances
instance Functor (MSt_res s) where
  fmap f (SReturn_st x) = SReturn_st (f x)
  fmap f (SBind_st n g) = SBind_st n (fmap f ∘ g)
instance Functor (MSt s) where
  fmap f m = SGetPut $ fmap (fmap (fmap f)) ∘ runMState m
joinMSt :: MSt s (MSt s a) → MSt s a
joinMSt m = SGetPut $ (=<<) magic ∘ runMState m
  where
  magic :: (Sem s, MSt_res s (MSt s a)) → MDec (Sem s, MSt_res s a)
  magic (s, SReturn_st m) = runMState m s
  magic (s, SBind_st n g) = Leaf (s, SBind_st n (joinMSt ∘ g))
instance Monad (MSt s) where
  return x = SGetPut $ λs → Leaf (s, SReturn_st x)
  m >>= f = joinMSt (fmap f m)
```

REFERENCES

[1] Andreas Abel and Christian Sattler. 2019. Normalization by Evaluation for Call-By-Push-Value and Polarized Lambda Calculus. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. 1–12.

[2] Danel Ahman and Sam Staton. 2013. Normalization by Evaluation and Algebraic Effects. In *MFPS (Electronic Notes in Theoretical Computer Science, Vol. 298)*. Elsevier, 51–69.

[3] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. 2001. Normalization by Evaluation for Typed Lambda Calculus with Coproducts. In *LICS*. IEEE Computer Society, 303–310.

[4] Markus Aronsson, Emil Axelsson, and Mary Sheeran. 2014. Stream processing for embedded domain specific languages. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*. 1–12.

[5] Robert Atkey, Sam Lindley, and Jeremy Yallop. 2009. Unembedding domain-specific languages. In *Haskell*. ACM, 37–48.

[6] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. 2010. The Design and Implementation of Feldspar - An Embedded Language for Digital Signal Processing. In *IFL (Lecture Notes in Computer Science, Vol. 6647)*. Springer, 121–136.

[7] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*. ACM, 64–76.

[8] Nick Benton and Andrew Kennedy. 2001. Exceptional syntax. *Journal of Functional Programming* 11, 4 (2001), 395–410.

[9] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalisation by Evaluation. In *Prospects for Hardware Foundations (Lecture Notes in Computer Science, Vol. 1546)*. Springer, 117–137.

[10] Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed lambda-calculus. In *LICS*. IEEE Computer Society, 203–211.

[11] Ilya Beylin and Peter Dybjer. 1995. Extracting a Proof of Coherence for Monoidal Categories from a Proof of Normalization for Monoids. In *TYPES (Lecture Notes in Computer Science, Vol. 1158)*. Springer, 47–61.

[12] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *ICFP*. ACM, 174–184.

[13] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543.

[14] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *DAMP*. ACM, 3–14.

[15] James Cheney, Sam Lindley, and Philip Wadler. 2013. A practical theory of language-integrated query. In *ICFP*. ACM, 403–416.

[16] Catarina Coquand. 1993. From Semantics to Rules: A Machine Assisted Analysis. In *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers (Lecture Notes in Computer Science, Vol. 832)*, Egon Börger, Yuri Gurevich, and Karl Meinke (Eds.). Springer, 91–105. https://doi.org/10.1007/BFb0049326

[17] Thierry Coquand and Peter Dybjer. 1997. Intuitionistic Model Constructions and Normalization Proofs. *Math. Struct. Comput. Sci.* 7, 1 (1997), 75–94.

[18] Olivier Danvy. 1998. Type-Directed Partial Evaluation. In *Partial Evaluation (Lecture Notes in Computer Science, Vol. 1706)*. Springer, 367–411.

[19] Olivier Danvy, Morten Rhiger, and Kristoffer Høgsbro Rose. 2001. Normalization by evaluation with typed abstract syntax. *J. Funct. Program.* 11, 6 (2001), 673–680.

[20] Peter Dybjer and Andrzej Filinski. 2000. Normalization and Partial Evaluation. In *APPSEM (Lecture Notes in Computer Science, Vol. 2395)*. Springer, 137–192.

[21] Andrzej Filinski. 1999. A Semantic Account of Type-Directed Partial Evaluation. In *PPDP (Lecture Notes in Computer Science, Vol. 1702)*. Springer, 378–395.

[22] Andrzej Filinski. 2001. Normalization by Evaluation for the Computational Lambda-Calculus. In *TLCA (Lecture Notes in Computer Science, Vol. 2044)*. Springer, 151–165.

[23] Neil Ghani. 1995. $\beta\eta$-Equality for Coproducts. In *TLCA (Lecture Notes in Computer Science, Vol. 902)*. Springer, 171–185.

[24] Andy Gill. 2014. Domain-specific languages and code synthesis using Haskell. *Commun. ACM* 57, 6 (2014), 42–49.

[25] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. 2010. Haskell Boards the Ferry - Database-Supported Program Execution for Haskell. In *IFL (Lecture Notes in Computer Science, Vol. 6647)*. Springer, 1–18.

[26] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (1996), 196.

[27] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., USA.

[28] Sam Lindley. 2005. *Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages.* Ph.D. Dissertation. University of Edinburgh.

[29] Sam Lindley. 2007. Extensional Rewriting with Sums. In *TLCA (Lecture Notes in Computer Science, Vol. 4583)*. Springer, 255–271.

[30] Sam Lindley. 2009. Accumulating bindings. In *NBE 2009*. 49–56.

[31] Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*. Vol. 80. Elsevier, 73–118.

[32] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Embedding by Normalisation. *CoRR* abs/1603.05197 (2016).

[33] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything old is new again: quoted domain-specific languages. In *PEPM*. ACM, 25–36.

[34] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *PLDI*. ACM, 199–208.

[35] Dag Prawitz. 1971. Ideas and results in proof theory. In *Proceedings of the 2nd Scandinavian Logic Symposium (Studies in Logics and the Foundations of Mathmatics, 63)*. North Holland, 235–307.

[36] Morten Rhiger. 2003. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.* 25, 3 (2003), 291–315.

[37] Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 2–9.

[38] Alejandro Russo, Koen Claessen, and John Hughes. 2008. A library for light-weight information-flow security in Haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008*. 13–24.

[39] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell,*. 95–106.

[40] Josef Svenningsson and Emil Axelsson. 2015. Combining deep and shallow embedding of domain-specific languages. *Comput. Lang. Syst. Struct.* 44 (2015), 143–165.

[41] Bo Joel Svensson, Mary Sheeran, and Ryan R. Newton. 2014. Design exploration through code-generating DSLs. *Commun. ACM* 57, 6 (2014), 56–63.

[42] Nachiappan Valliappan, Robert Krook, Alejandro Russo, and Koen Claessen. 2020. Towards secure IoT programming in Haskell. In *Haskell@ICFP*. ACM, 136–150.

[43] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. 2018. Partially-static data as free extension of algebras. *Proc. ACM Program. Lang.* 2, ICFP (2018), 100:1–100:30.

C

Simple Noninterference by Normalization

Abstract. Information-flow control (IFC) languages ensure programs preserve the confidentiality of sensitive data. *Noninterference*, the desired security property of such languages, states that public outputs of programs must not depend on sensitive inputs. In this paper, we show that noninterference can be proved using normalization. Unlike arbitrary terms, normal forms of programs are well-principled and obey useful syntactic properties—hence enabling a simpler proof of noninterference. Since our proof is syntax-directed, it offers an appealing alternative to traditional semantic based techniques to prove noninterference.

In particular, we prove noninterference for a static IFC calculus, based on Haskell's `seclib` library, using normalization. Our proof follows by straightforward induction on the structure of normal forms. We implement normalization using *normalization by evaluation* and prove that the generated normal forms preserve semantics. Our results have been verified in the Agda proof assistant.

1 INTRODUCTION

Information-flow control (IFC) is a security mechanism which guarantees confidentiality of sensitive data by controlling how information is allowed to flow in a program. The guarantee that programs secured by an IFC system do not leak sensitive data is often proved using a property called *noninterference*. Noninterference ensures that an observer authorized to view the output of a program (pessimistically called the attacker) cannot infer any sensitive data handled by it. For example, suppose that the type $\mathsf{Int}_{\mathrm{H}}$ denotes a secret integer and $\mathsf{Bool}_{\mathrm{L}}$ denotes a public boolean. Now consider a program f with the following type:

$$f : \mathsf{Int}_{\mathrm{H}} \rightarrow \mathsf{Bool}_{\mathrm{L}}$$

For this program, noninterference ensures that f outputs the same boolean for any given integer.

To prove noninterference, we must show that the public output of a program is not affected by varying the secret input. This has been achieved using many techniques including *term erasure* based on dynamic operational semantics [14, 23, 24, 29], denotational semantics [1, 13], and *parametricity* [27, 7, 3]. In this paper, we show that noninterference can also be proved by normalizing programs using the static or *residualising* semantics [15] of the language.

If a program returns the same output for any given input, it must be the case that it does not depend on the input to compute the output. Thus proving noninterference for a program which receives a secret input and produces a public output, amounts to showing that the program behaves like a *constant* program. For example, proving noninterference for the program f consists of showing that it is equivalent to either $\lambda\ x\,.\,\mathit{true}$ or $\lambda\ x\,.\,\mathit{false}$; it is immediately apparent that these functions do not depend on the secret input x. But how can we prove this for *any* arbitrary definition of f?

The program f may have been defined as the simple function $\lambda\ x\,.\,(\mathit{not}\ \mathit{false})$ or perhaps the more complex function $\lambda\ x\,.\,((\lambda\ y\,.\,\mathit{snd}\ (x\ ,\ y))\ \mathit{true})$. Observe, however, that both these programs can be normalized to the equivalent function $\lambda\ x\,.\,\mathit{true}$. In general, although terms in the language may be arbitrarily complex, their *normal forms* (such as $\lambda\ x\,.\,\mathit{true}$) are not. They are simpler, thus well-suited for showing noninterference.

The key idea in this paper is to normalize terms, and prove noninterference by simple structural induction on their normal forms. To illustrate this, we prove noninterference for a static IFC calculus, which we shall call λ_{sec}, based on Haskell's `seclib` library by Russo, Claessen, and Hughes We present the typing rules and static semantics for λ_{sec} by extending Moggi's *computational metalanguage* [19] (Section 2). We identify normal forms of λ_{sec}, and establish syntactic properties about a normal form's dependency on its input (Section 3). Using these properties, we show that the normal forms of program f are $\lambda\ x\,.\,\mathit{true}$ or $\lambda\ x\,.\,\mathit{false}$—as expected (Section 4).

To prove noninterference for all terms using normal forms, we implement normalization for λ_{sec} using *normalization by evaluation* (NbE) [6] and prove that it preserves the static semantics (Section 5). Using normalization, we prove noninterference for program f and further generalize this proof to *all* terms in λ_{sec} (Section 6) —including, for example, a program which operates on both secret and public values such as

$\mathsf{Bool}_\mathsf{L} \times \mathsf{Bool}_\mathsf{H} \to \mathsf{Bool}_\mathsf{L} \times \mathsf{Bool}_\mathsf{H}$. Finally, we conclude by discussing related work and future directions (Section 7).

Unlike earlier proofs, our proof shows that noninterference is an inherent property of the normal forms of λ_{sec}. Since the proof is primarily type and syntax-directed, it provides an appealing alternative to typical semantics based proof techniques. All the main theorems in this paper have been mechanized in the proof assistant Agda[1].

2 THE λ_{sec} CALCULUS

In this section we present λ_{sec}, a static IFC calculus that we shall use as the basis for our proof of noninterference. It models the pure and terminating fragment of the IFC library `seclib`[2] for Haskell, and is an extension of the calculus developed by Russo, Claessen, and Hughes [23] with sum types. `seclib` is a lightweight implementation of static IFC which allows programmers to incorporate untrusted third-party code into their applications while ensuring that it does not leak sensitive data. Below, we recall the public interface (API) of `seclib`:

$$\begin{aligned}
&\text{data } S\ (\ell :: \mathit{Lattice})\ a \\
&\mathit{return} :: a \to S\ \ell\ a \\
&(\gg\!=) :: S\ \ell\ a \to (a \to S\ \ell\ b) \to S\ \ell\ b \\
&\mathit{up} :: \ell_\mathsf{L} \sqsubseteq \ell_\mathsf{H} \Rightarrow S\ \ell_\mathsf{L}\ a \to S\ \ell_\mathsf{H}\ a
\end{aligned}$$

Similar to other IFC libraries in Haskell such as LIO [24] or MAC [30], `seclib`'s security guarantees rely on exposing the API to the programmer while hiding the underlying implementation. Programs written against the API and the *safe* parts of the language [25] are guaranteed to be *secure-by-construction*; the library enforces security statically through types. As an example, suppose that we have the two-point security lattice [11, see] {L, H} where the only disallowed flow is from secret (H) to public (L), denoted $\mathsf{H} \not\sqsubseteq \mathsf{L}$. The following program written using the `seclib` API is well-typed and—intuitively—secure:

$$\begin{aligned}
&\mathit{example} :: S\ \mathsf{L}\ \mathsf{Bool} \to S\ \mathsf{H}\ \mathsf{Bool} \\
&\mathit{example}\ p = \mathit{up}\ (p \gg\!= \lambda\ b \to \mathit{return}\ (\mathit{not}\ b))
\end{aligned}$$

The function *example* negates the Bool that it receives as input and upgrades its security level from public to secret. On the other hand, had the program tried to downgrade the secret input to public—clearly violating the policy of the security lattice—the typechecker would have rejected the program as ill-typed.

The Calculus. λ_{sec} is a simply typed λ-calculus (STLC) with a base (uninterpreted) type, unit type, product and sum types, and a security monad type for every security level in a set of labels (denoted by Label). The set of labels may be a lattice, but our development only requires it to be a preorder on the relation $\sqsubseteq$. Throughout the rest of this paper, we use the labels ℓ_L and ℓ_H and refer to them as *public* and *secret*, although they represent levels in an arbitrary security lattice such that $\ell_\mathsf{H} \not\sqsubseteq \ell_\mathsf{L}$. Figure 1 defines the syntax of terms, types and contexts of λ_{sec}.

[1] https://github.com/carlostome/ni-nbe
[2] https://hackage.haskell.org/package/seclib

$$
\begin{array}{lll}
\text{Label} & \ell\ ,\ \ell_{\mathrm{H}}\ ,\ \ell_{\mathrm{L}} & \\
\text{Context} & \Gamma\ \Delta\ \Sigma & ::= \emptyset \mid \Gamma\ ,\ x : \tau \\
\text{Type} & \tau\ \tau_1\ \tau_2 & ::= \tau_1 \Rightarrow \tau_2 \mid \iota \mid () \\
 & & \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \\
 & & \mid \mathsf{S}\ \ell\ \tau \\
\text{Term} & t\ s\ u & ::= x \mid \lambda\, x.t \mid t\ s \mid () \\
 & & \mid \langle t\ ,\ s \rangle \mid \mathsf{fst}\ t \mid \mathsf{snd}\ t \\
 & & \mid \mathsf{left}\ t \mid \mathsf{right}\ t \\
 & & \mid \mathsf{case}\ t\ (\mathsf{left}\ x_1 \to s)\ (\mathsf{right}\ x_2 \to u) \\
 & & \mid \mathsf{return}\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ u \mid \mathsf{up}\ t
\end{array}
$$

Fig. 1. The λ_{sec} calculus

In addition to the standard introduction and elimination constructs for unit, products and sums in STLC, λ_{sec} uses the constructs return, let and up for the security monad $\mathsf{S}\ \ell\ \tau$, which mirrors S from `seclib`. Note that our presentation favours let, as in Moggi [18], over the Haskell bind (≫=), although both presentations are equivalent—i.e. $t \gg\!= \lambda\, x.u$ can be encoded as $\mathsf{let}\ x = t\ \mathsf{in}\ u$.

The typing rules for return and let, shown in Figure 2, ensure that computations over labeled values in the security monad $\mathsf{S}\ \ell\ \tau$ do not leak sensitive data. The construct return allows the programmer to tag a value of type τ with security label ℓ; and bind enforces that sequences of computations over labeled values stay at the same security level.

$$\boxed{\Gamma \vdash t : \tau}$$

$$\textsc{Return}\ \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{return}\ t : \mathsf{S}\ \ell\ \tau} \qquad \textsc{Up}\ \frac{\Gamma \vdash t : \mathsf{S}\ \ell_{\mathrm{L}}\ \tau \qquad \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}}{\Gamma \vdash \mathsf{up}\ t : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau}$$

$$\textsc{Let}\ \frac{\Gamma \vdash t : \mathsf{S}\ \ell\ \tau_1 \qquad \Gamma\ ,\ x : \tau_1 \vdash s : \mathsf{S}\ \ell\ \tau_2}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ s : \mathsf{S}\ \ell\ \tau_2}$$

Fig. 2. Type system of λ_{sec} (excerpts)

Further, the calculus models the *up* combinator in `seclib` as the construct up. Its purpose is to relabel computations to higher security levels. The rule Up, shown in Figure 2, statically enforces that information can only flow from ℓ_{L} to ℓ_{H} in agreement with the security policy $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. The rest of the typing rules for λ_{sec} are standard [21], and thus omitted here. For a full account we refer the reader to our Agda formalization.

For completeness, the function *example* from earlier can be encoded in the λ_{sec} calculus as follows:[3]

[3]In λ_{sec}, the type Bool is encoded as () + () with *false* = left () and *true* = right ().

$$example \;=\; \lambda\, s.\mathsf{up}\ (\mathsf{let}\ b = s\ \mathsf{in}\ \mathsf{return}\ (not\ b))$$

Static Semantics. The static semantics of λ_{sec} is defined as a set of equations relating terms of the same type typed under the same environment. The equations characterize pairs of λ_{sec} terms that are equivalent based on β-reduction, η-expansion and other monadic operations.

We present the equations for return and let constructs of the monadic type S (à la Moggi [19]) in Figure 3, and further extend this with equations for the up primitive in Figure 4.

The remaining equations—including β and η rules for other types, and permutation rules for commuting case conversions—are fairly standard [15, 2], and can be found in the Agda formalization.

As customary, we use the notation $t_1\ [x/t\_2]$ for capture-avoiding substitution of the term t_2 for variable x in term t_1.

$$\boxed{\Gamma \vdash t_1 \approx t_2 : \tau}$$

$$\beta\text{-S}\quad \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma\,, x : \tau \vdash t_2 : \mathsf{S}\ \ell\ \tau}{\Gamma \vdash \mathsf{let}\ x = (\mathsf{return}\ t_1)\ \mathsf{in}\ t_2 \approx t_2\ [x/t\_1] : \mathsf{S}\ \ell\ \tau}$$

$$\eta\text{-S}\quad \frac{\Gamma \vdash t : \mathsf{S}\ \ell\ \tau}{\Gamma \vdash t \approx \mathsf{let}\ x = t\ \mathsf{in}\ (\mathsf{return}\ x) : \mathsf{S}\ \ell\ \tau}$$

$$\gamma\text{-S}\quad \frac{\Gamma \vdash t_1 : \mathsf{S}\ \ell\ \tau_1 \qquad \Gamma\,, x : \tau_1 \vdash t_2 : \mathsf{S}\ \ell\ \tau_2 \qquad \Gamma\,, x : \tau_1\,, y : \tau_2 \vdash t\_3 : \mathsf{S}\ \ell\ \tau_3}{\begin{array}{l}\Gamma \vdash \mathsf{let}\ x = (\mathsf{let}\ y = t_1\ \mathsf{in}\ t_2)\ \mathsf{in}\ t\_3 \approx \\ \mathsf{let}\ y = t_1\ \mathsf{in}\ (\mathsf{let}\ x = t_2\ \mathsf{in}\ t\_3) : \mathsf{S}\ \ell\ \tau_3\end{array}}$$

Fig. 3. Static semantics of λ_{sec} (return and let)

The up primitive induces equations regarding its interaction with itself and other constructs in the security monad. In Figure 4, we make the auxiliary condition of up and the label of return explicit using subscripts for better clarity. These equations can be understood as follows:

- Rule δ_1-S. applying up over let is equivalent to distributing it over the subterms of let.
- Rule δ_2-S. applying up on an term labeled as return t is equivalent to relabeling t with the final label.
- Rule δ_{trans}-S. applying up twice is equivalent to applying it once using the transitivity of the relation $\sqsubseteq$.
- Rule δ_{refl}-S. applying up using the reflexive relation $\ell \sqsubseteq \ell$ is equivalent to not applying it.

$$\boxed{\Gamma \vdash t_1 \approx t_2 : \tau}$$

$$\delta_1\text{-S}\ \frac{\Gamma \vdash t : \mathsf{S}\ \ell_{\mathrm{L}}\ \tau_1 \qquad \Gamma, x : \tau_1 \vdash u : \mathsf{S}\ \ell_{\mathrm{L}}\ \tau_2 \qquad p : \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}}{\Gamma \vdash \mathsf{up}_p\ (\mathsf{let}\ x = t\ \mathsf{in}\ u) \approx \mathsf{let}\ x = (\mathsf{up}_p\ t)\ \mathsf{in}\ (\mathsf{up}_p\ u) : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau}$$

$$\delta_2\text{-S}\ \frac{\Gamma \vdash t : \tau \qquad p : \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}}{\Gamma \vdash \mathsf{up}_p\ (\mathsf{return}_{\ell_{\mathrm{L}}}\ t) \approx \mathsf{return}_{\ell_{\mathrm{H}}}\ t : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau}$$

$$\delta_{\mathrm{TRANS}}\text{-S}\ \frac{\Gamma \vdash t : \mathsf{S}\ \ell_{\mathrm{L}}\ \tau \qquad p : \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{M}} \qquad q : \ell_{\mathrm{M}} \sqsubseteq \ell_{\mathrm{H}} \qquad r = \mathsf{trans}\text{-}\sqsubseteq\ p\ q}{\Gamma \vdash \mathsf{up}_q\ (\mathsf{up}_p\ t) \approx \mathsf{up}_r\ t : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau}$$

$$\delta_{\mathrm{REFL}}\text{-S}\ \frac{\Gamma \vdash t : \mathsf{S}\ \ell\ \tau \qquad p : \ell \sqsubseteq \ell}{\Gamma \vdash \mathsf{up}_p\ t \approx t : \mathsf{S}\ \ell\ \tau}$$

Fig. 4. Static semantics of λ_{sec} (up)

3 NORMAL FORMS OF λ_{sec}

As discussed in Section 1, our proof of noninterference utilizes syntactic properties of normal forms, and hence relies on normalizing terms in the language. Normal forms are a restricted subset of terms in the λ_{sec} calculus which intuitively corresponds to terms that cannot be normalized further. The syntax of normal forms is defined using two well-typed interdependent syntactic categories: *neutral* forms as $\Gamma \vdash_{\mathrm{ne}} t : \tau$ (Figure 5) and normal forms as $\Gamma \vdash_{\mathrm{nf}} t : \tau$ (Figure 6). Neutral forms are a special case of normal forms which depend entirely on the typing context (e.g., a variable).

Since the definition of neutral and normal forms are merely a syntactic restriction over terms, they can be embedded back into terms of λ_{sec} using a *quotation* function $\ulcorner n \urcorner$. This embedding can be implemented for neutrals and normal forms by simply mapping them to their term-counterparts.

$$\boxed{\Gamma \vdash_{\mathrm{ne}} t : \tau}$$

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash_{\mathrm{ne}} x : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash_{\mathrm{ne}} t : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash_{\mathrm{nf}} s : \tau_1}{\Gamma \vdash_{\mathrm{ne}} t\ s : \tau_2} \qquad \textsc{Fst}\ \frac{\Gamma \vdash_{\mathrm{ne}} t : \tau_1 \times \tau_2}{\Gamma \vdash_{\mathrm{ne}} \mathsf{fst}\ t : \tau_1}$$

$$\textsc{Snd}\ \frac{\Gamma \vdash_{\mathrm{ne}} t : \tau_1 \times \tau_2}{\Gamma \vdash_{\mathrm{ne}} \mathsf{snd}\ t : \tau_2}$$

Fig. 5. Neutral forms

$$\boxed{\Gamma \vdash_{\mathrm{nf}} t : \tau}$$

$$\textsc{Unit}\ \frac{}{\Gamma \vdash_{\mathrm{nf}} () : ()} \qquad \textsc{Lam}\ \frac{\Gamma\,,\, x : \tau_1 \vdash_{\mathrm{nf}} t : \tau_2}{\Gamma \vdash_{\mathrm{nf}} \lambda\, x.t : \tau_1 \Rightarrow \tau_2} \qquad \textsc{Base}\ \frac{\Gamma \vdash_{\mathrm{ne}} t : \iota}{\Gamma \vdash_{\mathrm{nf}} t : \iota} \qquad \textsc{Ret}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau}{\Gamma \vdash_{\mathrm{nf}} \mathsf{return}\ t : \mathsf{S}\ \ell\ \tau}$$

$$\textsc{LetUp}\ \frac{\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \qquad \Gamma \vdash_{\mathrm{ne}} t : \mathsf{S}\ \ell_{\mathrm{L}}\ \tau_1 \qquad \Gamma\,,\, x : \tau_1 \vdash_{\mathrm{nf}} s : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau_2}{\Gamma \vdash_{\mathrm{nf}} \mathsf{let}\uparrow\ x = t\ \mathsf{in}\ s : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau_2}$$

$$\textsc{Left}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau_1}{\Gamma \vdash_{\mathrm{nf}} \mathsf{left}\ t : \tau_1 + \tau_2} \qquad \textsc{Right}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau_2}{\Gamma \vdash_{\mathrm{nf}} \mathsf{right}\ t : \tau_1 + \tau_2}$$

$$\textsc{Case}\ \frac{\Gamma \vdash_{\mathrm{ne}} t : \tau_1 + \tau_2 \qquad \Gamma\,,\, x_1 : \tau_1 \vdash_{\mathrm{nf}} t_1 : \tau \qquad \Gamma\,,\, x_2 : \tau_2 \vdash_{\mathrm{nf}} t_2 : \tau}{\Gamma \vdash_{\mathrm{nf}} \mathsf{case}\ t\ (\mathsf{left}\ x_1 \longrightarrow t_1)\ (\mathsf{right}\ x_2 \longrightarrow t_2) : \tau}$$

Fig. 6. Normal forms

Neutral Forms. The neutral forms are terms which are characterized by a property called *neutrality*, which is stated as follows:

Property 3.1 (Neutrality)**.** For a given neutral form of type $\Gamma \vdash_{\mathrm{ne}} \tau$, neutrality states that the type τ must occur as a *subformula* of a type in the context Γ.

For instance, given a neutral form $\Gamma \vdash_{\mathrm{ne}} n : \mathsf{Bool}$, neutrality states that the type Bool must occur as a subformula of some type in the typing context Γ. An example of such a context is $\Gamma = [x : () \Rightarrow \mathsf{Bool}\ ,\ y : \mathsf{S}\ \ell_{\mathrm{H}}\ \iota]$. The notion of a subformula, originally defined for logical propositional formulas in proof theory [26], can also be defined for types as follows:

Definition 3.1 (Subformula)**.** For some types τ, τ_1 and τ_2; a subformula of a type is defined as:

- τ is a subformula of τ
- τ is a subformula of $\tau_1 \otimes \tau_2$ if τ is a subformula of τ_1 or τ is a subformula of τ_2, where $\otimes$ denotes the binary type operators $\times$, $+$ and $\Rightarrow$.

The type Bool occurs as a subformula in the typing context $[() \Rightarrow \mathsf{Bool}\ ,\ \mathsf{S}\ \ell_{\mathrm{H}}\ \iota]$ since the type Bool is a subformula of the type $() \Rightarrow \mathsf{Bool}$. Note, however, that the type ι does not occur as a subformula in this context since ι is not a subformula of the type $\mathsf{S}\ \ell_{\mathrm{H}}\ \iota$ by the above definition.

Normal Forms. Intuitively, normal forms of type $\Gamma \vdash_{\mathrm{nf}} \tau$ are characterized as terms of type $\Gamma \vdash \tau$ that cannot be *reduced* further using the static semantics. Precisely, a normal form is a term obtained by systematically applying the equations defined by the relation $\approx$ in a specific order to a given term. We leave the exact order of applying the equations unspecified since we only require that there *exists* a normal

form for every term—we prove this later in Section 5. The normal forms in Figure 6 extend the β-short η-long forms in STLC [5, 2] with return and let↑. Note that, unlike neutrals, arbitrary normal forms do not obey neutrality since they may also construct values which do not occur in the context. For example, the normal form left () (which denotes the value *false*) of type $\emptyset \vdash_{\text{nf}} \text{Bool}$ constructs a value of the type Bool in the empty context $\emptyset$.

The reader may have noticed that the let↑ construct in normal forms does not directly resemble a term, and hence it is not immediately obvious how it should be quoted. Normal forms constructed by let↑ can be quoted by first applying up to the quotation of the neutral and then using let. The reason let↑ represents both let and up in the normal forms is to retain the non-reducibility of normal forms. Had we added up separately to normal forms, then this may trigger further reductions. For example, the term up (return ()) can be reduced further to the term return (). Disallowing up-terms directly in normal forms removes the possibility of this reduction in normal forms. Similarly, adding up to neutral forms is also equally worse since it breaks neutrality.

The syntactic characterization of neutral and normal forms provides us with useful properties in the proof of noninterference. For example, there cannot exist a neutral of type $\emptyset \vdash_{\text{ne}} \tau$ for any type τ. By neutrality, if such a neutral form exists, then τ must be a subformula of the empty context $\emptyset$, but this is impossible! Similarly, the η-long form of normal forms guarantee that a normal form of a function type must begin with either a λ or case—hence reducing the number of possible cases in our proof. In the next section, we utilize these properties to show that the program f (from earlier) behaves as a constant.

4 NORMAL FORMS AND NONINTERFERENCE

The program $f : \text{Int}_{\text{H}} \to \text{Bool}_{\text{L}}$ from Section 1 can be generalized in λ_{sec} as a term[4] $\emptyset \vdash f : \text{S}\ \ell_{\text{H}}\ \tau \Rightarrow \text{S}\ \ell_{\text{L}}\ \text{Bool}$ marking the secret input and public output through the security monad. Noninterference for this term—which Russo, Claessen, and Hughes [23] refer to as a "noninterference-like" property for λ_{sec}—states that given two levels ℓ_{L} (*public*) and ℓ_{H} (*secret*) such that the flow of information from secret to public is disallowed as $\ell_{\text{H}} \not\sqsubseteq \ell_{\text{L}}$; for any two possibly different secrets *s_1* and *s_2*, applying f to *s_1* is equivalent to applying it to *s_2*. In other words, it states that varying the secret input must *not interfere* with the public output.

As explained before, for $\emptyset \vdash f : \text{S}\ \ell_{\text{H}}\ \tau \Rightarrow \text{S}\ \ell_{\text{L}}\ \text{Bool}$ to satisfy noninterference, it must be equivalent to the constant function whose body is return *true* or return *false* independent of the input. For an arbitrary program f it is not possible to conclude so just from case analysis—as programs may be fairly complex—however, for normal forms of the same type it is possible. In the lemma below, we materialize this intuition:

Lemma 4.1 (Normal forms of f are constant)**.** For any normal form $\emptyset \vdash_{\text{nf}} f : \text{S}\ \ell_{\text{H}}\ \tau \Rightarrow \text{S}\ \ell_{\text{L}}\ \text{Bool}$, either $f \equiv \lambda\, x\,.\,(\text{return}\ \mathit{true})$ or $f \equiv \lambda\, x\,.\,(\text{return}\ \mathit{false})$

Note that the equality relation $\equiv$ denotes syntactic (or propositional) equality, which means that the normal forms on both sides must be syntactically identical. The proof

[4] λ_{sec} does not have polymorphic types, in this case τ represents an arbitrary but concrete type, for instance unit ().

follows by direct case analysis on the normal forms of type $\emptyset \vdash_{\mathrm{nf}} f : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau \Rightarrow \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool}$:

Proof of Lemma 4.1. Upon closer inspection of the normal forms of λ_{sec} (Figure 6), the reader may notice that for the function type $\emptyset \vdash_{\mathrm{nf}} \mathsf{S}\ \ell_{\mathrm{H}}\ \tau \Rightarrow \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool}$ there exists only two possibilities: a case or a λ construct. The former, can be easily dismissed by neutrality because it requires the scrutinee—a neutral form of sum type $\tau_1 + \tau_2$—to appear in the empty context. In the latter case, the λ construct extends typing context of the body with the type of the argument, and thus refines the normal form to have the shape $\lambda\, x\,.\, \_$ where $\emptyset\ ,\ x : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau \vdash_{\mathrm{nf}} \_ : \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool}$.

Considering the normal forms of type $\emptyset\ ,\ x : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau \vdash_{\mathrm{nf}} \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool}$, we realize that there are only three possible candidates: the case construct again, the monadic return or let. As before, case is discharged because it requires the scrutinee of sum type to occur in the context $\emptyset\ ,\ x : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau$. Analogously, the monadic let with a neutral term of type $\mathsf{S}\ \ell_{\mathrm{L}}\ \tau$, expects this type to occur in the same context—but it does not, since $\mathsf{S}\ \ell_{\mathrm{L}}\ \tau$ is not a subformula of $\mathsf{S}\ \ell_{\mathrm{H}}\ \tau$. The remaining case, return, can be further refined, where the only possibilities leave us with $\lambda\, x\,.\, (\mathsf{return}\ \mathit{true})$ or $\lambda\, x\,.\, (\mathsf{return}\ \mathit{false})$. □

In order to show that noninterference holds for arbitrary programs of type $\emptyset \vdash f : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau \Rightarrow \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool}$ using this lemma, we must link the behaviour of a program with that of its normal form. In the next section we develop the necessary normalization machinery and later complete the proof of noninterference in Section 6.

5 FROM λ_{sec} TO NORMAL FORMS

The goal of this section is to implement a normalization algorithm that bridges the gap between terms and their normal forms. For this purpose, we employ Normalization by Evaluation (NbE).

Normalization based on rewriting techniques [21] perform syntactic transformations of a term to produce a normal form. NbE, on the other hand, normalizes a term by evaluating it in a host language, and then extracting a normal form from the (semantic) value in the host language. Evaluation of a term is implemented by an interpreter function eval, and the extraction of normal forms, called *reification*, is implemented by an inverse function reify. Normalization is implemented as a function from terms to normal forms by composing these functions:

$$\begin{aligned}&\mathsf{norm} : (\Gamma \vdash \tau) \to (\Gamma \vdash_{\mathrm{nf}} \tau)\\ &\mathsf{norm}\ t = \mathsf{reify}\ (\mathsf{eval}\ t)\end{aligned}$$

The function eval and reify have the following types in the host language:

$$\begin{aligned}&\mathsf{eval} : (\Gamma \vdash \tau) \to (\llbracket\, \Gamma \,\rrbracket \to \llbracket\, \tau \,\rrbracket)\\ &\mathsf{reify} : (\llbracket\, \Gamma \,\rrbracket \to \llbracket\, \tau \,\rrbracket) \to (\Gamma \vdash_{\mathrm{nf}} \tau)\end{aligned}$$

In these types, the function $\llbracket\ \_\ \rrbracket$ interprets types and contexts in λ_{sec} as types in the host language. That is, the type $\llbracket\ \tau\ \rrbracket$ denotes the interpretation of the (λ_{sec}) type τ in the host language, and similarly for $\llbracket\ \Gamma\ \rrbracket$. On the other hand, the function $\llbracket\ \Gamma\ \rrbracket \to \llbracket\ \tau\ \rrbracket$—a function between the interpretations in the host language—denotes the interpretation of the term $\Gamma \vdash \tau$.

The advantages of using NbE over a rewrite system are two-fold: first, it serves as an actual implementation of the normalization algorithm; second, and most importantly, when implemented in a proof system like Agda, it makes normalization amenable to formal reasoning. For example, since Agda ensures that all functions are total, we are assured that a normal form must exist for every term in λ_{sec}. Similarly, we also get a proof that normalization terminates for free since Agda ensures that all functions are terminating.

We implement the functions eval and reify for terms in λ_{sec} using Agda as the host language. Note that, however, the implementation of our algorithm—and NbE in general—is not specific to Agda. It may also be implemented in other programming languages such as Haskell [10] or Standard ML [5].

In the remainder of this section, we will denote the typing derivations $\Gamma \vdash_{\text{nf}} \tau$ and $\Gamma \vdash_{\text{ne}} \tau$ as Nf τ and Ne τ respectively. We leave the context Γ implicit to avoid the clutter caused by contexts and their *weakenings* [4, 16]. Similarly, we will represent variables of type $\tau \in \Gamma$ as Var τ, leaving Γ implicit. Although we use de Bruijn indices in the actual implementation of variables, we will continue to use named variables here to ease presentation. We encourage the curious reader to see the formalization in Agda for further details.

5.1 NbE for Simple Types

To begin with, we implement evaluation and reification for the types ι, $()$, $\times$ and $\Rightarrow$. The implementation for sums is more technical, and hence deferred to Appendix A. Note that the implementation of NbE for simple types is entirely standard [4, 5]. Their interpretation as Agda types is defined as follows:

$$
\begin{aligned}
[\![\ \iota\ ]\!] &= \text{Nf}\ \iota \\
[\![\ ()\ ]\!] &= \top \\
[\![\ \tau_1 \times \tau_2\ ]\!] &= [\![\ \tau_1\ ]\!] \times [\![\ \tau_2\ ]\!] \\
[\![\ \tau_1 \Rightarrow \tau_2\ ]\!] &= [\![\ \tau_1\ ]\!] \to [\![\ \tau_2\ ]\!]
\end{aligned}
$$

The types $()$, $\times$ and $\Rightarrow$ are simply interpreted as their counterparts in Agda. For the base type ι, however, we cannot provide a counterpart in Agda since we do not know anything about this type. Instead, since the type ι is not constructed or eliminated by any specific construct in λ_{sec}, we simply require a normal form as an evidence for producing a value of type ι—and thus interpret it as Nf ι.

Typing contexts map variables to types, and hence their interpretation is an execution environment (or equivalently, a semantic substitution) defined like-wise:

$$
\begin{aligned}
[\![\ \emptyset\ ]\!] &= \emptyset \\
[\![\ \Gamma\ ,\ x : \tau_1\ ]\!] &= [\![\ \Gamma\ ]\!]\ [\ \text{Var}\ \tau_1 \mapsto [\![\ \tau_1\ ]\!]\ ]
\end{aligned}
$$

For example, a value γ which inhabits the interpretation $[\![\ \Gamma\ ]\!]$ denotes the execution environment for evaluating a term typed in the context Γ.

Given these definitions, evaluation is implemented as a straightforward interpreter function:

$$
\begin{aligned}
\text{eval}\ x \quad \gamma &= \text{lookup}\ x\ \gamma \\
\text{eval}\ () \quad \gamma &= \text{tt}
\end{aligned}
$$

```
eval (fst t)       γ = π₁ (eval t γ)
eval (snd t)       γ = π₂ (eval t γ)
eval (< t₁ , t₂ >) γ = (eval t₁ γ , eval t₂ γ)
eval (λ x.t)       γ = λ v → eval t (γ [x ↦ v])
eval (t s)         γ = (eval t γ) (eval s γ)
```

Note that γ is an execution environment for the term's context; lookup, π_1 and π_2 are Agda functions; and tt is the constructor of the unit type $\top$. For the case of $\lambda\ x.t$, evaluation is expected to return an equivalent semantic function. We compute the body of this function by evaluating the body term t using the substitution γ extended with a mapping which assigns the value v to the variable x—denoted $\gamma\ [x \mapsto v]$.

Reification, on the other hand, is implemented using two helper functions reflect and reifyVal. The function reflect converts neutral forms to semantic values, while the dual function reifyVal converts semantic values to normal forms. These functions are implemented as follows:

```
reifyVal : ⟦ τ ⟧ → Nf τ
reifyVal {ι} n          = n
reifyVal {()} tt        = ()
reifyVal {τ₁ × τ₂} p    =
  < reifyVal {τ₁} (π₁ p) , reifyVal {τ₂} (π₁ p) >
reifyVal {τ₁ ⇒ τ₂} f    =
  λ x.reifyVal {τ₂} (f (reflect {τ₁} x)) | fresh x

reflect : Ne τ → ⟦ τ ⟧
reflect {ι} n           = n
reflect {()} n          = tt
reflect {τ₁ × τ₂} n     =
  (reflect {τ₁} (fst n) , reflect {τ₂} (snd n))
reflect {τ₁ ⇒ τ₂} n     =
  λ v → reflect {τ₂} (n (reifyVal {τ₁} v))
```

Note that the argument inside the braces { } denotes an implicit parameter, which is the type of the corresponding neutral/value argument of reflect/reifyVal here.

Reflection is implemented by performing a type-directed translation of neutral forms to semantic values by induction on types. The interpretation of types, defined earlier, guides our implementation. For example, reflection of a neutral with a function type must produce a function value since the type $\Rightarrow$ is interpreted as an Agda function. For this purpose, we are given the argument value in the semantics and it remains to construct a function body of the appropriate type. We produce the body of this function by recursively reflecting a neutral application of the function and (the reification of) the argument value. The function reifyVal is also implemented in a similar fashion by induction on types.

To implement reification, recollect that the argument to reify is a function that results from partially applying the eval function with a term. If the term has type $\Gamma \vdash \tau$, then the argument, say f, must have the type $[\![\ \Gamma\ ]\!] \to [\![\ \tau\ ]\!]$. Thus, to apply

f, we need an execution environment of the type $[\![\ \Gamma\ ]\!]$. This environment can be generated by simply reflecting the variables in the context as follows:

```
genEnv : (Γ : Ctx)  →  [[ Γ ]]
genEnv ∅            =  ∅
genEnv (Γ , x : τ)  =  genEnv Γ [ x ↦ reflect x ]
```

Finally, we can now implement reify as follows:

```
reify {Γ} f  =  let γ  =  genEnv Γ in reifyVal (f γ)
```

We generate an environment γ to apply the semantic function f, and then convert the resulting semantic value to a normal form by applying reifyVal.

5.2 NbE for the Security Monad

To interpret a type $S\ \ell\ \tau$, we need a semantic counterpart in the host language which is also a monad. Suppose that we define such a monad as an inductive data type T parameterized by a label ℓ and some type a (which would be $[\![\ \tau\ ]\!]$ in this case). Evidently this monad must allow the implementation of the semantic counterparts of the terms return, let and up in λ_{sec} as follows:

```
return : a  →  T ℓ a
bind   : T ℓ a  →  (a  →  T ℓ b)  →  T ℓ b
up     : (ℓL ⊑ ℓH)  →  T ℓL a  →  T ℓH a
```

To satisfy this specification, we define the data type T in Agda with the following constructors:

$$\textsc{Return}\ \frac{x : a}{\mathsf{return}\ x : T\ \ell\ a} \qquad \textsc{BindN}\ \frac{p : \ell_L \sqsubseteq \ell_H \qquad n : \mathsf{Ne}\ S\ \ell_L\ \tau \qquad f : \mathsf{Var}\ \tau \to T\ \ell_H\ a}{\mathsf{bindNe}\ p\ n\ f : T\ \ell_H\ a}$$

The constructor return returns a semantic value in the monad, while bindNe registers a binding of a neutral to monadic value. These constructors are the semantic equivalent of return and let↑ in the normal forms, respectively. The constructor bindNe is more general than the required function bind in order to allow the definition of up, which is defined by induction as follows:

```
up p (return v)  =
  return v
up p (bindNe q n f)  =
  bindNe (trans q p) n (λ x  →  up p (f x))
```

To understand this implementation, suppose that $p : \ell_M \sqsubseteq \ell_H$ for some labels ℓ_M and ℓ_H. A monadic value of type $T\ \ell_M\ a$ which is constructed by a return can be simply re-labeled to $T\ \ell_H\ a$ since return can be used to construct a monadic value on any label. For the case of bindNe $q\ n\ f$, we have that $q : \ell_L \sqsubseteq \ell_M$ and $n : \mathsf{Ne}\ S\ \ell_L\ \tau_1$, hence $\ell_L \sqsubseteq \ell_H$ by transitivity, and we may simply use bindNe to register n and recursively apply up on the continuation f to produce the desired result of type $T\ \ell_H\ a$.

Using the type T in the host language, we may now interpret the monad in λ_{sec} as follows:

$$[\![\, \mathsf{S}\ \ell\ \tau \,]\!] \;=\; T\ \ell\ [\![\, \tau \,]\!]$$

Having mirrored the monadic primitives in λ_{sec} using semantic counterparts, evaluation is rather simple:

```
eval (return t) γ  =  return (eval t γ)
eval (up p t)   γ  =  up p (eval t γ)
eval (let x = t in s) γ =
  bind (eval t γ) (λ v → eval s (γ [x ↦ v]))
```

For implementing reflection, we can use bindNe to register a neutral binding and recursively reflect the given variable:

```
reflect {S ℓ τ} n =
  bindNe refl n (λ x → return (reflect {τ} x))
```

Since we do not need to increase the sensitivity of the neutral to bind it here, we simply provide the "reflexive flow" $\mathsf{refl} : \ell \sqsubseteq \ell$.

The function reifyVal, on the other hand, is rather straightforward since the constructors of T are essentially semantic counterparts of the normal forms, and can hence be translated to it:

```
reifyVal {S ℓ τ} (return v) =
  return (reifyVal {τ} v)
reifyVal {S ℓ τ} (bindNe {p} n f) =
  let↑ {p} x = n in reifyVal {τ} (f x)
```

5.3 Preservation of Semantics

To prove that normalization preserves static semantics of λ_{sec}, we must show that the normal form of term is equivalent to the term. Since normal forms and terms belong to different syntactic categories, we must first quote normal forms to state this relationship using the term equivalence relation $\approx$. This property, called *consistency* of normal forms, is stated as follows:

Theorem 5.1 (Consistency of normal forms)**.** For any term $\Gamma \vdash t : \tau$ we have that $\Gamma \vdash t \approx \ulcorner\, \mathsf{norm}\ t \,\urcorner : \tau$

An attempt to prove consistency by induction on the terms or types fails quickly since the induction principle alone is not strong enough to prove this theorem. To solve this issue we must establish a notion of equivalence between a term and its interpretation using *logical relations* [22]. Using these relations, we can prove that evaluation is consistent by showing that it is *related* to applying a substitution in the syntax. Following this, we can also prove the consistency of reification by showing that reifying a value related to a term, yields a normal form which is equivalent to the term when quoted. The consistency of evaluation and reification yields the proof of consistency for normal forms.

This proof follows the style of the consistency proof of NbE for STLC using Kripke logical relations by Coquand [8]. As is the case for sums, NbE for the security monad uses an inductively defined data type to implement the semantic monad. Hence, we are able to leverage the proof techniques used to prove the consistency of NbE

for sums [28] to prove the same for the security monad. We skip the details of the proof here, but encourage the curious reader to see the Agda mechanization of this theorem.

6 NONINTERFERENCE FOR λ_{sec}

After developing the necessary machinery to normalize terms in the calculus, we are ready to state and prove noninterference for λ_{sec}. First, we complete the proof of noninterference for the program f from Section 4.

6.1 Special Case of Noninterference

Theorem 6.1 (Noninterference for f)**.** Given security levels ℓ_{L} and ℓ_{H} such that $\ell_{\text{H}} \not\sqsubseteq \ell_{\text{L}}$ and a function $\emptyset \vdash f : \mathsf{S}\ \ell_{\text{H}}\ \tau \Rightarrow \mathsf{S}\ \ell_{\text{L}}\ \mathsf{Bool}$ then $\forall\ s\_1\ s\_2 : \mathsf{S}\ \ell_{\text{H}}\ \tau.\ f\ s\_1 \approx f\ s\_2$

The proof of Theorem 6.1 relies upon two key ingredients: Lemma 4.1 (Section 4), which characterizes the shape of the normal forms of f; and consistency of normal forms, Theorem 5.1 (Section 5.3), which links the semantics of f with that of its normal forms.

Proof of Theorem 6.1. To show that a function $\emptyset \vdash f : \mathsf{S}\ \ell_{\text{H}}\ \tau \Rightarrow \mathsf{S}\ \ell_{\text{L}}\ \mathsf{Bool}$ is equivalent when applied to two different secret inputs s_1 and s_2, first, we instantiate Lemma 4.1 with the normal form of f, denoted by $\mathsf{norm}\ f$. In this manner, we obtain that the normal forms of f are exactly the constant function that returns *true* or *false* wrapped in the return. In the former case, by correctness of normalization we have that $f \approx \ulcorner\ \mathsf{norm}\ f\ \urcorner \approx \lambda\ x\,.\,\mathsf{return}\ \mathit{true}$. By β-reduction and congruence of term-level function application, we have that $\forall\ t.\ (\lambda\ x\,.\,\mathsf{return}\ \mathit{true})\ t \approx \mathsf{return}\ \mathit{true}$. Therefore, $f\ s\_1 \approx f\ s\_2$. The case when $\mathsf{norm}\ f \equiv \lambda\ x\,.\,\mathsf{return}\ \mathit{false}$ follows a similar argument. □

The noninterference property proven above characterizes what it means for a concrete class of programs, i.e. those of type $\emptyset \vdash f : \mathsf{S}\ \ell_{\text{H}}\ \tau \Rightarrow \mathsf{S}\ \ell_{\text{L}}\ \mathsf{Bool}$, to be secure: the attacker cannot even learn one bit of the secret from using program f. Albeit interesting, this property does not scale to more complex programs; for instance if the function f was typed in a non empty context the proof of the above lemma would not hold. The rest of this section is dedicated to generalize and prove noninterference from the program f to arbitrary programs written in λ_{sec}. As will become clear, normal forms of λ_{sec} play a crucial role towards proving noninterference.

6.2 General Noninterference Theorem

In order to discuss general noninterference for λ_{sec}, we must first specify what are the *secret* (ℓ_{H}) inputs of a program and its *public* (ℓ_{L}) output with respect to an attacker at level ℓ_{L}. The attacker can only learn information of a program by running it with different secret inputs and then observing its public output. Because the attacker can only observe outputs at their security level, we restrict the security condition to only consider programs where outputs are fully observable, i.e., *transparent* and *ground*, to the attacker.

Definition 6.1 (Transparent type)**.**

- $()$ is transparent at any level ℓ.
- ι is transparent at any level ℓ.
- $\tau_1 \Rightarrow \tau_2$ is transparent at ℓ iff τ_2 is transparent at ℓ.
- $\tau_1 + \tau_2$ is transparent at ℓ iff τ_1 and τ_2 are transparent at ℓ.
- $\tau_1 \times \tau_2$ is transparent at ℓ iff τ_1 and τ_2 are transparent at ℓ.
- $\mathsf{S}\ \ell'\ \tau$ is transparent at ℓ iff $\ell' \sqsubseteq \ell$ and τ is transparent at ℓ.

Definition 6.2 (Ground type)**.**

- $()$ is ground.
- ι is ground.
- $\tau_1 + \tau_2$ is ground iff τ_1 and τ_2 are ground.
- $\tau_1 \times \tau_2$ is ground iff τ_1 and τ_2 are ground.
- $\mathsf{S}\ \ell\ \tau$ is ground iff τ is ground.

A type τ is transparent at security level ℓ_{L} if the type does not include the security monad type over a higher security level ℓ_{H}. A ground type, on the other hand, is a first order type, i.e, a type that does not contain a function type. These simplifying restrictions over the output type of a program allow us to state a generic noninterference property over terms and perform induction on the normal forms.

These restrictions do not hinder the generality of our security condition: a program producing a partially public output, for instance a product $\mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool} \times \mathsf{S}\ \ell_{\mathrm{H}}\ \mathsf{Bool}$, can be transformed to produce a fully public output by applying the snd projection. We return to this example later at the end of the section. Also note that previous work on proving noninterference for static IFC languages [1, 17] impose similar restrictions.

Departing from the traditional view of programs as closed terms, i.e. terms without free variables, in the λ_{sec} calculus we consider all terms for which a typing derivation exists. This includes terms that contain free variables—unknowns—typed by the context, which we identify as the program inputs. Note that open terms are more general since they can always be closed as a function by abstracting over the free variables.

Now, we state what it means for a context to be secret at level ℓ. These definitions, dubbed ℓ-sensitivity, force the types appearing in the context to be at least as sensitive as ℓ.

Definition 6.3 (Context sensitivity)**.**

A context Γ is ℓ-sensitive if and only if for all types $\tau \in \Gamma$, τ is ℓ-sensitive. A type τ is ℓ-sensitive, on the other hand, if and only if:

- τ is the function type $\tau_1 \Rightarrow \tau_2$ and τ_2 is ℓ-sensitive.
- τ is the product type $\tau_1 \times \tau_2$ and τ_1 and τ_2 are ℓ-sensitive.
- τ is the monadic type $\mathsf{S}\ \ell'\ \tau_1$ and $\ell \sqsubseteq \ell'$.

Next, we define substitutions[5], which lay at the core of β-reduction rules in the λ_{sec} calculus. Substitutions map free variables in a term to other terms possibly typed in a different context.

[5] In Section 2 we purposely left capture-avoiding substitutions underspecified, we amend that here.

$$\textit{Substitution}\ \sigma ::= \sigma_\emptyset \mid \sigma\ [x \mapsto t]$$

$$\boxed{\Gamma \vdash_{\text{sub}} \sigma : \Delta}$$

$$\frac{\Gamma \vdash_{\text{sub}} \sigma : \Delta \qquad \Gamma \vdash t : \tau}{\Gamma \vdash_{\text{sub}} \sigma\ [\ x \mapsto t\ ] : \Delta\ ,\ x : \tau} \qquad \frac{}{\Gamma \vdash_{\text{sub}} \sigma_\emptyset : \emptyset}$$

Fig. 7. Substitutions for λ_{sec}

A substitution is either empty, $\sigma_\emptyset$, or is the substitution σ extended with a new mapping from the variable $x : \tau$ to term t. We denote $t\ [\ \sigma\ ]$ the application of substitution σ to term t. Its definition is standard by induction on the term structure, thus we omit it here and refer the reader to the Agda formalization.

Substitutions, in general, provide a mix of terms of secret and public type to fill the variables in the context Γ of a program. However, for noninterference we need to fix the public part of the substitution and allow the secret part to vary. We do so by splitting a substitution σ into the composition of a public substitution, $\Gamma \vdash_{\text{sub}} \sigma_{\ell_L} : \Delta$, that fixes the public inputs, and a secret substitution $\Delta \vdash_{\text{sub}} \sigma_{\ell_H} : \Sigma$, that restricts Δ to be ℓ_H-sensitive. The composition of both, denoted $\Gamma \vdash_{\text{sub}} (\sigma_{\ell_L}\ ;\ \sigma_{\ell_H}) : \Sigma$, maps variables in context Γ to terms typed in Σ: first, σ_{ℓ_L} maps variables from Γ to terms in Δ, subsequently, σ_{ℓ_H} maps variables in Δ to terms typed in Σ. Below, we state ℓ_L-equivalence of substitutions:

Definition 6.4 (Low equivalence of substitutions)**.**
Two substitutions σ_1 and σ_2 are ℓ_L-equivalent , written $\sigma_1 \approx_{\ell_L} \sigma_2$, if and only if for all ℓ_H such that $\ell_H \not\sqsubseteq \ell_L$, there exists a public substitution σ_{ℓ_L}, and two secret substitutions $\sigma^1_{\ell_H}$ and $\sigma^2_{\ell_H}$, such that $\sigma_1 \equiv \sigma_{\ell_L}\ ;\ \sigma^1_{\ell_H}$ and $\sigma_2 \equiv \sigma_{\ell_L}\ ;\ \sigma^2_{\ell_H}$

Informally, noninterference for λ_{sec} states that applying two low equivalent substitutions to an arbitrary term whose type is ground and transparent yields two equivalent programs. As previously explained, intuitively a program satisfies such property if it is equivalent to a *constant* program: i.e. a program where the output does not depend on the input—in this case the variables in the typing context. As in Section 4, instead of defining and proving this on arbitrary terms, we achieve this using normal forms.

Constant Terms and Normal Forms. We prove the noninterference theorem by showing that terms of a type at level ℓ_L, typed in a ℓ_H-sensitive context, must be constant. We achieve this in turn by showing that the normal forms of such terms are constant. Below, we state when a term is constant:

Definition 6.5 (Constant term)**.**
A term $\Gamma \vdash t : \tau$ is said to be constant if, for any two substitutions σ_1 and σ_2, we have that $t\ [\ \sigma_1\ ] \approx t\ [\ \sigma_2\ ]$.

Similarly, we must define what it means for a normal form to be constant. However, we cannot state this for normal forms directly using substitutions since the result of applying a substitution to a normal form may not be a normal form. For example, the result of substituting the variable x in the normal form $x : \iota \Rightarrow \iota\ ,\ y : \iota \vdash_{\text{nf}} x\ y : \iota$

by the identity function is not a normal form—and *cannot* be derived syntactically as a normal form using $\vdash_{\text{nf}}$. Instead, we lean on the shape of the context to state the property.

If a normal form $\Gamma \vdash_{\text{nf}} n : \tau$ is constant, then there must exist a syntactically identical derivation $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $n \equiv n'$. However, since n and n' are typed in different contexts, Γ and $\emptyset$, it is not possible to compare them for syntactic equality. We solve this problem by *renaming* the normal form n' to add as many variables as mentioned in context Γ. The signature of the renaming function is the following:

$$\text{ren} : \{\Gamma \leqslant \Delta\} \to (\Gamma \vdash_{\text{nf}} \tau) \to (\Delta \vdash_{\text{nf}} \tau)$$

The relation $\leqslant$ between contexts Γ and Δ indicates that the variables appearing in Δ are at least those present in Γ. This relation, called *weakening*, is defined as follows:

- $\emptyset \leqslant \emptyset$
- If $\Gamma \leqslant \Delta$, then $\Gamma \leqslant \Delta , x : \tau$
- If $\Gamma \leqslant \Delta$, then $\Gamma , x : \tau \leqslant \Delta , x : \tau$

The function ren can be defined by simple induction on the derivation of the normal forms. Note that terms can also be renamed in the same fashion.

Definition 6.6 (Constant normal form)**.** A normal form $\Gamma \vdash_{\text{nf}} n : \tau$ is constant if there exists a normal form $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $\text{ren}\ (n') \equiv n$.

Further, we need a lemma showing that if a term is constant, then so is its normal form.

Lemma 6.2 (Constant plumbing lemma)**.** If the normal form n of a term $\Gamma \vdash t : \tau$ is constant, then so is t.

The proof follows by induction on the normal forms:

Proof of Lemma 6.2. If n is constant, then there must exist a normal form $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $\text{ren}\ (n') \equiv n$. Let the quotation of this normal form $\ulcorner\ n'\ \urcorner$ be some term $\emptyset \vdash t' : \tau$. Recall from earlier that terms can also be renamed, hence we have $\text{ren}\ (t') \approx \text{ren}\ (\ulcorner\ n'\ \urcorner)$ by correctness of n'. Since it can be shown that $\text{ren}\ (\ulcorner\ n'\ \urcorner) \equiv \ulcorner\ \text{ren}\ (n')\ \urcorner$, we have that $\text{ren}\ (\ulcorner\ n'\ \urcorner) \equiv \ulcorner\ n\ \urcorner$, and by correctness of n, we also have $\text{ren}\ (t') \approx t$ — (1).

A substitution σ maps free variables in a term to terms. The empty substitution, denoted $\sigma_\emptyset$, is the unique substitution, such that $\Delta \vdash t'\ [\ \sigma_\emptyset\ ] : \tau$ for any Δ. That is, applying the empty substitution simply renames the term. We can show that $t'\ [\ \sigma_\emptyset\ ] \equiv \text{ren}\ (t')$, and hence, by (1), we have $t'\ [\ \sigma_\emptyset\ ] \approx t$ — (2). Since $\sigma_\emptyset$ renames a term typed in the empty context, we can show that for any substitution σ, we have $(t'\ [\ \sigma_\emptyset\ ])\ [\ \sigma\ ] \approx t'\ [\ \sigma_\emptyset\ ]$. Because $\sigma_\emptyset$ is also unique, for any two substitutions σ_1 and σ_2, we have $(t'\ [\ \sigma_\emptyset\ ])\ [\ \sigma_1\ ] \approx (t'\ [\ \sigma_\emptyset\ ])\ [\ \sigma_2\ ]$ by transitivity of $\approx$. As a result, from (2), we achieve the desired result, $t\ [\ \sigma_1\ ] \approx t\ [\ \sigma_2\ ]$, therefore t must be constant. □

The key insight of our noninterference proof is reflected in the following lemma which shows how normal forms of λ_{sec} typed in a sensitive context are either constant or the flow between the security level of the context and the output type is permitted.

Below we include the proof to showcase how it follows by straightforward induction on the shape of the normal forms.

Lemma 6.3 (Normal forms do not leak)**.** Given a normal form $\Gamma \vdash_{\mathrm{nf}} n : \tau$, where the context Γ is ℓ_i-sensitive, and τ is a ground and transparent type at level ℓ_o, then either n is constant or $\ell_i \sqsubseteq \ell_o$.

Proof. By induction on the structure of the normal form n. Note that λ and case normal forms need not be considered since the preconditions ensure that τ cannot be a function type (dismisses λ), and Γ cannot contain a variable of a sum type (dismisses case).

- **Case 1** ($\Gamma \vdash_{\mathrm{nf}} () : ()$). The normal form $()$ is constant.
- **Case 2** ($\Gamma \vdash_{\mathrm{nf}} n : \iota$). In this case, we are given the neutral n by the [Base] rule in Figure 6. It can be shown by induction that for all neutrals of type $\Gamma \vdash_{\mathrm{ne}} \tau$, if Γ is ℓ_i-sensitive and τ is transparent at ℓ_o, then $\ell_i \sqsubseteq \ell_o$. Hence, n gives us that $\ell_i \sqsubseteq \ell_o$.
- **Case 3** ($\Gamma \vdash_{\mathrm{nf}} \mathsf{return}\ n : \mathsf{S}\ \ell\ \tau$). By applying the induction hypothesis on the normal form n, we have that n is either constant or $\ell_i \sqsubseteq \ell_o$. In the latter case, we are done since we already have $\ell_i \sqsubseteq \ell_o$. In the former case, there exists a normal form n' such that $\mathsf{ren}\ (n') \equiv n$. By congruence of the relation $\equiv$, we get that $\mathsf{return}\ (\mathsf{ren}\ (n')) \equiv \mathsf{return}\ n$. Note that the function ren is defined as $\mathsf{ren}\ (\mathsf{return}\ n') \equiv \mathsf{return}\ (\mathsf{ren}\ n')$, and hence by transitivity of $\equiv$, we have that
  $\mathsf{ren}\ (\mathsf{return}\ (n')) \equiv \mathsf{return}\ n$. Thus, the normal form $\mathsf{return}\ n$ is also constant.
- **Case 4** ($\Gamma \vdash_{\mathrm{nf}} \mathsf{let}{\uparrow}\ x = n\ \mathsf{in}\ m : \mathsf{S}\ \ell_2\ \tau_2$). For this case, we have a neutral $\Gamma \vdash_{\mathrm{ne}} n : \mathsf{S}\ \ell_1\ \tau_1$ such that $\ell_1 \sqsubseteq \ell_2$, by the [LetUp] rule in Figure 6. Similar to case 2, we have that $\ell_i \sqsubseteq \ell_1$ from the neutral n. Hence, $\ell_i \sqsubseteq \ell_2$ by transitivity of the relation $\sqsubseteq$. Additionally, since $\mathsf{S}\ \ell_2\ \tau$ is transparent at ℓ_o, it must be the case that $\ell_2 \sqsubseteq \ell_o$ by definition of transparency. Therefore, once again by transitivity, we have $\ell_i \sqsubseteq \ell_o$.
- **Case 5** ($\Gamma \vdash_{\mathrm{nf}} \mathsf{left}\ n : \tau_1 + \tau_2$). Similar to return.
- **Case 6** ($\Gamma \vdash_{\mathrm{nf}} \mathsf{right}\ n : \tau_1 + \tau_2$). Similar to return.

□

The last step to noninterference is an ancillary lemma which shows that terms typed in ℓ_{H}-sensitive contexts are constant:

Lemma 6.4. Given a term $\Gamma \vdash t : \tau$, where the context Γ is ℓ_{H}-sensitive, and τ is a ground type transparent at ℓ_{L}. If $\ell_{\mathrm{H}} \not\sqsubseteq \ell_{\mathrm{L}}$, then t is constant.

The proof follows from lemmas Lemma 6.3 and Lemma 6.2.

Finally, we are ready to formally state and prove the noninterference property for programs written in λ_{sec}, which effectively demonstrates that programs do not leak sensitive information. The proof follows from the previous lemmas, which characterize the behaviour of programs by the syntactic properties of their normal forms.

Theorem 6.5 (Noninterference for λ_{sec})**.** Given security levels ℓ_{L} and ℓ_{H} such that $\ell_{\mathrm{H}} \not\sqsubseteq \ell_{\mathrm{L}}$; an attacker at level ℓ_{L}; two ℓ_{L}-*equivalent* substitutions σ_1 and σ_2 such that

$\sigma_1 \approx_{\ell_L} \sigma_2$; and a type τ that is ground and transparent at ℓ_L; then for any term $\Gamma \vdash t : \tau$ we have that $t\ [\ \sigma_1\ ] \approx t\ [\ \sigma_2\ ]$.

Proof of Theorem 6.5. Low equivalence of substitutions $\sigma_1 \approx_{\ell_L} \sigma_2$ gives that $\sigma_1 = \sigma_{\ell_L}\ ;\ \sigma^1_{\ell_H}$ and $\sigma_2 = \sigma_{\ell_L}\ ;\ \sigma^2_{\ell_H}$. After applying the public substitution σ_{ℓ_L} to the term $\Gamma \vdash t : \tau$, we are left with a term typed in a ℓ_H-sensitive context $\Delta, \Delta \vdash t\ [\ \sigma_{\ell_L}\ ] : \tau$. By Lemma 6.4, $t\ [\ \sigma_{\ell_L}\ ]$ is constant which means that $(t\ [\ \sigma_{\ell_L}\ ])\ [\ \sigma^1_{\ell_H}\ ] \approx (t\ [\ \sigma_{\ell_L}\ ])\ [\ \sigma^2_{\ell_H}\ ]$. By readjusting substitutions using composition we obtain $t\ ([\ \sigma_{\ell_L}\ ;\ \sigma^1_{\ell_H}\ ]) \approx t\ ([\ \sigma_{\ell_L}\ ;\ \sigma^2_{\ell_H}\ ])$, which yields $t\ [\ \sigma_1\ ] \approx t\ [\ \sigma_2\ ]$. □

6.3 Follow-up Example

To conclude this section, we briefly show how to instantiate the theorem of noninterference for λ_{sec} for programs of type $\emptyset \vdash t : \mathsf{S}\ \ell_L\ \mathsf{Bool} \times \mathsf{S}\ \ell_H\ \mathsf{Bool} \Rightarrow \mathsf{S}\ \ell_L\ \mathsf{Bool} \times \mathsf{S}\ \ell_H\ \mathsf{Bool}$, which are the recurring example for explaining noninterference in the literature [23, 7]. Adapted to the notion of noninterference based on substitutions, the corollary we aim to prove is the following:

Corollary 6.6 (Noninterference for t)**.** Given security levels ℓ_L and ℓ_H such that $\ell_H \not\sqsubseteq \ell_L$ and a program $x : \mathsf{S}\ \ell_L\ \mathsf{Bool} \times \mathsf{S}\ \ell_H\ \mathsf{Bool} \vdash t : \mathsf{S}\ \ell_L\ \mathsf{Bool} \times \mathsf{S}\ \ell_H\ \mathsf{Bool}$ then $\forall\ p : \mathsf{S}\ \ell_L\ \mathsf{Bool}\ ,\ s\_1\ s\_2 : \mathsf{S}\ \ell_H\ \mathsf{Bool}$. we have that $t\ [\ x \mapsto (p\ ,\ s\_1)\ ] \approx t\ [\ x \mapsto (p\ ,\ s\_2)\ ]$.

Because the main noninterference theorem requires the output to be fully observable by the attacker, we transform t to the desired shape by applying the snd projection. This is justified because the first component of the output is protected at level ℓ_H, which the attacker cannot observe. Below we prove noninterference for $x : \mathsf{S}\ \ell_L\ \mathsf{Bool} \times \mathsf{S}\ \ell_H\ \mathsf{Bool} \vdash \mathsf{snd}\ t : \mathsf{S}\ \ell_H\ \mathsf{Bool}$:

Proof of Corollary 6.6. To apply Theorem 6.5 we have to show that both substitutions are low equivalent, $[\ x \mapsto (p\ ,\ s\_1)\ ] \approx_{\ell_L} [\ x \mapsto (p\ ,\ s\_2)\ ]$ The key idea is that the substitution $[\ x \mapsto (p\ ,\ s\_1)\ ]$ can be decomposed into a public substitution $\sigma_{\ell_L} \equiv [\ x \mapsto (p\ ,\ y)\ ]$ and two different secret substitutions where each replaces the variable y by a different secret, $\sigma^1_{\ell_H} \equiv [\ y \mapsto s\_1\ ]$ and $\sigma^2_{\ell_H} \equiv [\ y \mapsto s\_2\ ]$. Now, the proof follows directly from Theorem 6.5. □

7 CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel proof of noninterference for the λ_{sec} calculus (based on Haskell's IFC library `seclib`) using normalization. The simplicity of the proof relies upon the normal forms of the calculus, which as opposed to arbitrary terms, are well-principled. To obtain normal forms from terms, we have implemented normalization using NbE, and shown that normal forms obey useful syntactic-properties such as neutrality and $\beta\eta$-long form. Most of the auxiliary lemmas and definitions towards proving noninterference build on these properties. Because normal forms are well-principled, many cases of the proofs follow directly by structural induction.

An important difference between our work and previous proofs based on term erasure is that our proof utilizes the static semantics of the language instead of

the dynamic semantics. Specifically, our proof of noninterference is not tied to any particular evaluation strategy, such as call-by-name or call-by-value, assuming the strategy is adequate with respect to the static semantics.

Perhaps the closest to our line of work is the proof of noninterference by Miyamoto and Igarashi [17] for a modal lambda calculus using normalization. The main novelty of our proof is that it works for standard extensions of the simply typed lambda calculus and does not change the typing rules of the underlying calculus (as presented and implemented by Russo, Claessen, and Hughes [23]). This makes our proof technique applicable even in the presence of other useful normalization-preserving extensions of STLC. For example, it should be possible to extend our proof for λ_{sec} further with exceptions and other *computational* effects (à la Moggi [18]) since our security monad is already an instance of this. Moreover, our proof relies on syntactic properties of normal forms in an open typing context since normalization is based on the static semantics of the language.

In this work we have only considered a calculus which models terminating computations. This opens up a question of whether our proof technique is applicable to languages which support general recursion, where computations need not necessarily terminate. The extensibility of this technique to recursion relies directly upon the choice of static semantics for normalizing recursion. For example, it may be possible to extend the proof for λ_{sec} with a fix-point combinator by treating it as an uninterpreted constant during normalization. That is, it may be sufficient to normalize the body of the function by ignoring the recursive application, because if the body does not leak a secret, then its recursive call must not either. Since complete normalization is not strictly needed for our purposes, we believe that our technique can also be extended to general recursion.

Our NbE implementation for λ_{sec} extends NbE for Moggi's computational metalanguage [12, 15] with a family of monads parameterized by a pre-ordered set of labels. This resembles the parameterization of monads by effects specified by a pre-ordered monoid, also known as *graded monads* [31, 20], and thus indicates the extensibility of our NbE algorithm to calculi with graded monads. It would be interesting to see if our proof technique can be used to prove noninterference for static enforcement of IFC using graded monads.

Using static semantics means that our work lays a foundation for static analysis of noninterference-like security properties. This opens up a plethora of exciting opportunities for future work. For example, one possibility would be to use type-direction partial evaluation [9] to simplify programs and inspect the resulting programs to verify if they violate security properties. Another arena would be the extension of our proof to more expressive IFC calculi such as DCC or MAC [30]. The main challenge here would be to identify the appropriate static semantics of the language, as they may not always have been designed with one in mind.

ACKNOWLEDGMENTS

We thank Alejandro Russo, Fabian Ruch, Sandro Stucki and Maximilian Algehed for the insightful discussions on normalization and noninterference. We would also like to thank Irene Lobo Valbuena, Claudio Agustin Mista and the anonymous reviewers at PLAS'19 for their comments on earlier drafts of this paper. This work was funded

by the Swedish Foundation for Strategic Research (SSF) under the projects WebSec (Ref. RIT17-0011) and Octopi (Ref. RIT17-0023).

REFERENCES

[1] Martín Abadi et al. "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 147–160. DOI: 10.1145/292540.292555. URL: https://doi.org/10.1145/292540.292555.

[2] Andreas Abel and Christian Sattler. "Normalization by Evaluation for Call-By-Push-Value and Polarized Lambda Calculus". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 2019, 3:1–3:12. DOI: 10.1145/3354166.3354168. URL: https://doi.org/10.1145/3354166.3354168.

[3] Maximilian Algehed. "A Perspective on the Dependency Core Calculus". In: *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by Mário S. Alvim and Stéphanie Delaune. ACM, 2018, pp. 24–28. DOI: 10.1145/3264820.3264823. URL: https://doi.org/10.1145/3264820.3264823.

[4] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. "Categorical Reconstruction of a Reduction Free Normalization Proof". In: *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*. Ed. by David H. Pitt, David E. Rydeheard, and Peter T. Johnstone. Vol. 953. Lecture Notes in Computer Science. Springer, 1995, pp. 182–199. DOI: 10.1007/3-540-60164-3_27. URL: https://doi.org/10.1007/3-540-60164-3%5C_27.

[5] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. "Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 64–76. DOI: 10.1145/964001.964007. URL: https://doi.org/10.1145/964001.964007.

[6] Ulrich Berger and Helmut Schwichtenberg. "An Inverse of the Evaluation Functional for Typed lambda-calculus". In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, 1991, pp. 203–211. DOI: 10.1109/LICS.1991.151645. URL: https://doi.org/10.1109/LICS.1991.151645.

[7] William J. Bowman and Amal Ahmed. "Noninterference for free". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 101–113. DOI: 10.1145/2784731.2784733. URL: https://doi.org/10.1145/2784731.2784733.

[8] Catarina Coquand. "From Semantics to Rules: A Machine Assisted Analysis". In: *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers*. Ed. by Egon Börger, Yuri Gurevich, and

Karl Meinke. Vol. 832. Lecture Notes in Computer Science. Springer, 1993, pp. 91–105. DOI: 10.1007/BFb0049326. URL: https://doi.org/10.1007/BFb0049326.

[9] Olivier Danvy. "Type-Directed Partial Evaluation". In: *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*. Ed. by John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann. Vol. 1706. Lecture Notes in Computer Science. Springer, 1998, pp. 367–411. DOI: 10.1007/3-540-47018-2_16. URL: https://doi.org/10.1007/3-540-47018-2%5C_16.

[10] Olivier Danvy, Morten Rhiger, and Kristoffer Høgsbro Rose. "Normalization by evaluation with typed abstract syntax". In: *J. Funct. Program.* 11.6 (2001), pp. 673–680. DOI: 10.1017/S0956796801004166. URL: https://doi.org/10.1017/S0956796801004166.

[11] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5 (1976), pp. 236–243. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056.

[12] Andrzej Filinski. "Normalization by Evaluation for the Computational Lambda-Calculus". In: *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*. Ed. by Samson Abramsky. Vol. 2044. Lecture Notes in Computer Science. Springer, 2001, pp. 151–165. DOI: 10.1007/3-540-45413-6_15. URL: https://doi.org/10.1007/3-540-45413-6%5C_15.

[13] G. A. Kavvos. "Modalities, cohesion, and information flow". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 20:1–20:29. DOI: 10.1145/3290333. URL: https://doi.org/10.1145/3290333.

[14] Peng Li and Steve Zdancewic. "Arrows for secure information flow". In: *Theor. Comput. Sci.* 411.19 (2010), pp. 1974–1994. DOI: 10.1016/j.tcs.2010.01.025. URL: https://doi.org/10.1016/j.tcs.2010.01.025.

[15] Sam Lindley. "Normalisation by evaluation in the compilation of typed functional programming languages". PhD thesis. University of Edinburgh, UK, 2005. URL: http://hdl.handle.net/1842/778.

[16] Conor McBride. "Everybody's Got To Be Somewhere". In: *Proceedings of the 7th Workshop on Mathematically Structured Functional Programming, MSFP@FSCD 2018, Oxford, UK, 8th July 2018*. Ed. by Robert Atkey and Sam Lindley. Vol. 275. EPTCS. 2018, pp. 53–69. DOI: 10.4204/EPTCS.275.6. URL: https://doi.org/10.4204/EPTCS.275.6.

[17] Kenji Miyamoto and Atsushi Igarashi. "A modal foundation for secure information flow". In: *In Proceedings of IEEE Foundations of Computer Security (FCS)*. 2004, pp. 187–203.

[18] 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*. IEEE Computer Society, 1989, pp. 14–23. DOI: 10.1109/LICS.1989.39155. URL: https://doi.org/10.1109/LICS.1989.39155.

[19] Eugenio Moggi. "Notions of Computation and Monads". In: *Inf. Comput.* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4.

[20] Dominic A. Orchard and Tomas Petricek. "Embedding effect systems in Haskell". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, 2014, pp. 13–24. DOI: 10.1145/2633357.2633368. URL: https://doi.org/10.1145/2633357.2633368.

[21] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

[22] Gordon D. Plotkin. "Lambda-definability in the full type hierarchy". In: *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*. Academic Press, London-New York, 1980, pp. 363–373.

[23] Alejandro Russo, Koen Claessen, and John Hughes. "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, 2008, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289.

[24] Deian Stefan et al. "Flexible dynamic information flow control in Haskell". In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. Ed. by Koen Claessen. ACM, 2011, pp. 95–106. DOI: 10.1145/2034675.2034688. URL: https://doi.org/10.1145/2034675.2034688.

[25] David Terei et al. "Safe haskell". In: *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*. Ed. by Janis Voigtländer. ACM, 2012, pp. 137–148. DOI: 10.1145/2364506.2364524. URL: https://doi.org/10.1145/2364506.2364524.

[26] Anne Sjerp Troelstra and Helmut Schwichtenberg. *Basic proof theory, Second Edition*. Vol. 43. Cambridge tracts in theoretical computer science. Cambridge University Press, 2000. ISBN: 978-0-521-77911-1.

[27] Stephen Tse and Steve Zdancewic. "Translating dependency into parametricity". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. Ed. by Chris Okasaki and Kathleen Fisher. ACM, 2004, pp. 115–125. DOI: 10.1145/1016850.1016868. URL: https://doi.org/10.1145/1016850.1016868.

[28] Nachiappan Valliappan and Alejandro Russo. "Exponential Elimination for Bicartesian Closed Categorical Combinators". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 2019, 20:1–20:13. DOI: 10.1145/3354166.3354185. URL: https://doi.org/10.1145/3354166.3354185.

[29] Marco Vassena and Alejandro Russo. "On Formalizing Information-Flow Control Libraries". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*. Ed. by Toby C. Murray and Deian Stefan. ACM, 2016, pp. 15–28. DOI: 10.1145/2993600.2993608. URL: https://doi.org/10.1145/2993600.2993608.

[30] Marco Vassena et al. "MAC A verified static information-flow control library". In: *Journal of Logical and Algebraic Methods in Programming* 95 (2018), pp. 148–180. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S235222081730069X.

[31] Philip Wadler and Peter Thiemann. "The marriage of effects and monads". In: *ACM Trans. Comput. Log.* 4.1 (2003), pp. 1–32. DOI: 10.1145/601775.601776. URL: https://doi.org/10.1145/601775.601776.

A NBE FOR SUMS

It is tempting to interpret sums component-wise like products and functions as: $[\![\tau_1 + \tau_2]\!] = [\![\tau_1]\!] \uplus [\![\tau_2]\!]$. However, this interpretation makes it impossible to implement reflection faithfully: should the reflection of a variable $x : \tau_1 + \tau_2$ be a semantic value of type $[\![\tau_1]\!]$ (left injection) or $[\![\tau_2]\!]$ (right injection)? We cannot make this decision since the value which substitutes x may be either of these cases. The standard solution to this issue is to interpret sums using *decision trees* [2]. A decision tree allows us to defer this decision until more information is available about the injection of the actual value.

As in the previous case for the monadic type T, a decision tree can be defined as an inductive data type D parameterized by some type interpretation a with the following constructors:

$$\textsc{Leaf}\ \frac{x : a}{\mathsf{leaf}\ x : D\ a} \qquad \textsc{Branch}\ \frac{n : \mathsf{Ne}\ (\tau_1 + \tau_2) \qquad f : \mathsf{Var}\ \tau_1 \to D\ a \qquad g : \mathsf{Var}\ \tau_2 \to D\ a}{\mathsf{branch}\ n\ f\ g : D\ a}$$

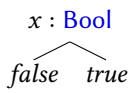
The leaf constructor constructs a leaf of the tree from a semantic value, while the branch constructor constructs a tree which represents a suspended decision over the value of a sum type. The branch constructor is the semantic equivalent of case in normal forms.

Decision trees allow us to model semantic sum values, and hence allow the interpretation of the sum type as follows:

$$[\![\tau_1 + \tau_2]\!] = D\ ([\![\tau_1]\!] \uplus [\![\tau_2]\!])$$

We interpret a sum type (in λ_{sec}) as a decision tree which contains a value of the sum type (in Agda).

As an example, the term *false* of type Bool, implemented as left (), will be interpreted as a decision tree leaf (inj_1 tt) of type $D\ [\![\mathsf{Bool}]\!]$ since we know the exact injection. The Agda constructor inj_1 denotes the left injection in Agda, and inj_2 the right injection. For a variable x of type Bool, however, we cannot interpret it as a leaf since we don't know the actual injection that may substitute it. Instead, it is interpreted as a decision tree by branching over the possible values as branch x ($\lambda\ \_ \to$ leaf (inj_1 tt)) ($\lambda\ \_ \to$ leaf (inj_2 tt))[6]—which intuitively represents the following tree:



In light of this interpretation of sums, the implementation of evaluation for injections is straightforward since we only need to wrap the appropriate injection inside a leaf:

[6]We ignore the argument (as $\lambda\ \_$) here since it has the uninteresting type ()

```
eval (left t)  γ = leaf (inj₁ (eval t γ))
eval (right t) γ = leaf (inj₂ (eval t γ))
```

For evaluating case however, we must first implement a decision procedure since case is used to make a choice over sums.

To make a decision over a tree of type $D\ [\![\ \tau\ ]\!]$, we need a function mkDec : $D\ [\![\ \tau\ ]\!] \to [\![\ \tau\ ]\!]$. It can be implemented by induction on the type τ using monadic functions fmap and join on trees, which can in turn be implemented by straightforward structural induction on the tree. Additionally, we will also need a function which converts a decision over normal forms to a normal form: convert : $D\ (\mathsf{Nf}\ \tau) \to \mathsf{Nf}\ \tau$. The implementation of this function is made possible by the fact that branch resembles case in normal forms, and can hence be translated to it. We skip the implementation of these functions here, but encourage the reader to see the Agda implementation.

Using these definitions, we can now complete evaluation as follows:

```
eval (case t (left x₁ → t₁) (right x₂ → t₂)) γ =
  mkDec (fmap match (eval t γ))
  where
    match : ([[ τ₁ ]] ⊎ [[ τ₂ ]]) → [[ τ ]]
    match (inj₁ v) = eval t₁ (γ [x_1 ↦ v])
    match (inj₂ v) = eval t₂ (γ [x_2 ↦ v])
```

We first evaluate the term t of type $\tau_1 + \tau_2$ to obtain a tree of type $D\ ([\![\ \tau_1\ ]\!] \uplus [\![\ \tau_2\ ]\!])$. Then, we map the function match which eliminates the sum inside the decision tree to $[\![\ \tau\ ]\!]$, to produce a tree of type $D\ [\![\ \tau\ ]\!]$. Finally, we run the decision procedure mkDec on the resulting decision tree to produce the desired value of type $[\![\ \tau\ ]\!]$.

Reflection for a neutral of a sum type can now be implemented using branch as follows:

```
reflect {τ₁ + τ₂} n =
  branch n
    (leaf (λ x₁ → inj₁ (reflect {τ₁} x₁)))
    (leaf (λ x₂ → inj₂ (reflect {τ₂} x₂)))
```

As discussed earlier, we construct the decision tree for neutral n using branch. The subtrees represent all possible semantic values of n and are constructed by reflecting the variables x_1 and x_2.

The function reifyVal, on the other hand, is implemented similar to evaluation by eliminating the sum value inside the decision tree into normal forms as follows:

```
reifyVal {τ₁ + τ₂} tr = convert (fmap matchNf tr)
  where
    matchNf : ([[ τ₁ ]] + [[ τ₂ ]]) → Nf (τ₁ + τ₂)
    matchNf (inj₁ x) = left (reifyVal {τ₁} x)
    matchNf (inj₂ y) = right (reifyVal {τ₂} y)
```

With this function, we have completed the implementation of NbE for sums.

D

Exponential Elimination for Bicartesian Closed Categorical Combinators

Abstract. Categorical combinators offer a simpler alternative to typed lambda calculi for static analysis and implementation. Since categorical combinators are accompanied by a rich set of conversion rules which arise from categorical laws, they also offer a plethora of opportunities for program optimization. It is unclear, however, how such rules can be applied in a systematic manner to eliminate intermediate values such as *exponentials*, the categorical equivalent of higher-order functions, from a program built using combinators. Exponential elimination simplifies static analysis and enables a simple closure-free implementation of categorical combinators—reasons for which it has been sought after.

In this paper, we prove exponential elimination for *bicartesian closed* categorical (BCC) combinators using normalization. We achieve this by showing that BCC terms can be normalized to normal forms which obey a weak subformula property. We implement normalization using Normalization by Evaluation, and also show that the generated normal forms are correct using logical relations.

1 INTRODUCTION

Categorical combinators are combinators designed after arrows, or *morphisms*, in category theory. Although originally introduced to present the connection between lambda calculus and cartesian closed categories (CCCs) [13], categorical combinators have attracted plenty of attention in formal analysis and implementation of various lambda calculi. For example, they are commonly used to formulate an evaluation model based on abstract machines [12, 16]. Abadi et al. [1] observe that categorical combinators "make it easy to derive machines for the λ-calculus and to show the correctness of these machines". This ease is attributed to the absence of variables in combinators, which avoids the difficulty with variable names, typing contexts, substitution, etc. Recently, categorical combinators have also been used in practical applications for programming *smart contracts* on the blockchain [24] and compiling functional programs [14].

Since categorical combinators are based on categorical models, they are accompanied by a rich set of *conversion* rules (between combinator terms) which emerge from the equivalence between morphisms in the model. These conversion rules form the basis for various correct program transformations and optimizations. For example, Elliott [14] uses conversion rules from CCCs to design various rewrite rules to optimize the compilation of Haskell programs to CCC combinators. The availability of these rules raises a natural question for optimizing terms in categorical combinator languages: can intermediate values be eliminated by applying the conversion rules whenever possible?

The ability to eliminate intermediate values in a categorical combinator language has plenty of useful consequences, just as in functional programming. For example, the elimination of *exponentials*, the equivalent of high-order functions, from BCC combinators solves problems created by exponentials in static analysis [27], and has also been sought after for interpreting functional programs in categories without exponentials ([14], Section 10.2). It has been shown that normalization erases higher-order functions from a program with first-order input and output types in the simply typed lambda calculus (STLC) with products and sums [22]—also known as *defunctionalization* [25]. Similarly, can we erase exponentials and other intermediate values by normalizing programs in the equally expressive *bicartesian closed* categorical (BCC) combinators?

We implement normalization for BCC combinators towards eliminating intermediate values, and show that it yields exponential elimination. We first recall the term language and conversion rules for BCC combinators (Section 2), and provide a brief overview of the normalization procedure (Section 3). Then, we identify normal forms of BCC terms which obey a *weak subformula* property and prove exponential elimination by showing that these normal forms can be translated to an equivalent first-order combinator language without exponentials (Section 4 and Section 5).

To assign a normal form to every term in the language, we implement a normalization procedure using *Normalization by Evaluation* (NbE) [8, 9] (Section 6). We then prove, using *Kripke logical relations* [20], that normal forms of terms are consistent with the conversion rules by showing that they are inter-convertible. (Section 7). Furthermore, we show that exponential elimination can be used to simplify static

analysis—while retaining expressiveness—of a combinator language called Simplicity (Section 8). Finally, we conclude by discussing related work (Section 9) and final remarks (Section 10).

Although we only discuss the elimination of exponentials in this chapter, the elimination of intermediate values of other types can also be achieved likewise—except for *products*. The reason for this becomes apparent when we discuss the weak subformula property (in Section 5.1).

We implement normalization and mechanize the correctness proof in the dependently-typed language Agda [10, 23]. This chapter is also written in literate Agda since dependent types provide a uniform framework for discussing both programs and proofs. We use category theoretic terminology to organize the implementation based on the categorical account of NbE by Altenkirch et al. [4]. However, all the definitions, algorithms, and proofs here are written in vanilla Agda, and the reader may view them as regular programming artifacts. Hence, we do not require that the reader be familiar with advanced categorical concepts. We discuss the important parts of the implementation here, and encourage the curious reader to see the complete implementation[1] for further details.

2 BCC COMBINATORS

A BCC combinator has an input and an output type, which can be one of the following: $\mathbf{1}$ (for unit), $\mathbf{0}$ (for empty), $*$ (for product), $+$ (for sum), $\Rightarrow$ (for exponential) and base (for base types). The Agda data type BCC (see Figure 1) defines the term language for BCC combinators. In the definition, the type Ty denotes a BCC type, and Set denotes a type definition in Agda (like $*$ in Haskell). Note that the type variables a, b and c are implicitly quantified and hidden here. The combinators are self-explanatory and behave like their functional counterparts. Unlike functions, however, these combinators do not have a notion of variables or typing contexts.

```
data BCC : Ty → Ty → Set where
  id     : BCC a a
  _•_    : BCC b c → BCC a b → BCC a c
  unit   : BCC a 1
  init   : BCC 0 a
  exl    : BCC (a * b) a
  exr    : BCC (a * b) b
  pair   : BCC a b → BCC a c → BCC a (b * c)
  inl    : BCC a (a + b)
  inr    : BCC b (a + b)
  match  : BCC a c → BCC b c → BCC (a + b) c
  curry  : BCC (c * a) b → BCC c (a ⇒ b)
  apply  : BCC (a ⇒ b * a) b
```

Fig. 1. BCC Combinators

[1] https://github.com/nachivpn/expelim

The BCC combinators are accompanied by a number of conversion rules which emerge from the equational theory of bicartesian closed categories [6]. These rules can be formalized as an equivalence relation `_≈_ : BCC a b → BCC a b → Set` (see Figure 2). In the spirit of categorical laws, the type-specific conversion rules can be broadly classified as *elimination* and *uniqueness* (or *universality*) rules. The elimination rules state when the composition of two terms can be eliminated, and uniqueness rules state the unique structure of a term for a certain type. For example, the conversion rules for products include two elimination rules (exl-pair, exr-pair) and a uniqueness rule (uniq-pair):

Note that the operator ⊗ used in the exponential elimination rule (apply-curry) is defined below. It pairs two BCC terms using pair and applies them on each component of a product. The components are projected using exl and exr respectively.

```
_⊗_ : BCC a b → BCC c d → BCC (a ∗ c) (b ∗ d)
f ⊗ g = pair (f • exl) (g • exr)
```

The standard $\beta\eta$ conversion rules of STLC [3, 7] can be derived from the conversion rules specified here. This suggests that we can perform β and η conversion for BCC terms, and normalize them as in STLC. Let us look at a few simple examples.

Example 1. For a term `f : BCC a (b ∗ c)`, `pair (exl • f) (exr • f)` can be converted to `f` as follows.

```
eta∗ : {f : BCC a (b ∗ c)} → pair (exl • f ) (exr • f ) ≈ f
eta∗ = uniq-pair refl refl
```

The constructor refl states that the relation ≈ is reflexive. The conversion above corresponds to η conversion for products in STLC.

Example 2. Suppose that we define a combinator uncurry as follows.

```
uncurry : BCC a (b ⇒ c) → BCC (a ∗ b) c
uncurry f = apply • f ⊗ id
```

Given this definition, a term `curry (uncurry f)` can be converted to `f`, by unfolding the definition of uncurry—as `curry (apply • f ⊗ id)`—and then using uncurry refl.

```
eta⇒ : {f : BCC a (b ⇒ c)} → curry (uncurry f ) ≈ f
eta⇒ = uniq-curry refl
```

Note that Agda unfolds the definition of uncurry automatically for us. The conversion above corresponds to η conversion for functions in STLC.

Example 3. Given a term `t : BCC a (b ∗ c)` such that `t ≈ (pair f g) • h : BCC a (b ∗ c)`, t can be converted to the term `pair (f • h) (g • h)` using equational reasoning such as the following.

```
    t
  ≈ (pair f g) • h                          By definition
  ≈ pair (exl • pair f g • h) (exr • pair f g • h)   By example 1
  ≈ pair (f • h) (exr • pair f g •h)        By exl-pair
  ≈ pair (f • h) (g • h)                    By exr-pair
```

```
data _≈_ : BCC a b → BCC a b → Set where
  - categorical rules
  idr           :{f : BCC a b}
    → f • id ≈ f
  idl           : {f : BCC a b}
    → id • f ≈ f
  assoc         : {h : BCC a b} {g : BCC b c} {f : BCC c d}
    → f • (g • h) ≈ f • (g • h)
  - elimination rules
  exl-pair      : {f : BCC c a} {g : BCC c b}
    → (exl • pair f g) ≈ f
  exr-pair      : {f : BCC c a} {g : BCC c b}
    → (exr • pair f g) ≈ g
  match-inl     : {f : BCC a c} {g : BCC b c}
    → (match f g • inl) ≈ f
  match-inr     : {f : BCC a c} {g : BCC b c}
    → (match f g • inr) ≈ g
  apply-curry : {f : BCC (a * b) c}
    → apply • (curry f ⊗ id) ≈ f
  - uniqueness rules
  uniq-init     : {f : BCC 0 a}
    → init ≈ f
  uniq-unit     : {f : BCC a 1}
    → unit ≈ f
  uniq-pair     : ∀ {f g} {h : BCC z (a * b)}
    → exl • h ≈ f → exr • h ≈ g → pair f g ≈ h
  uniq-curry    : {h : BCC a (b ⇒ c)} {f : BCC (a * b) c}
    → apply • h ⊗ id ≈ f → curry f ≈ h
  uniq-match : ∀ {f g} {h : BCC (a + b) z}
    → h • inl ≈ f → h • inr ≈ g → match f g ≈ h
  - equivalence and congruence rules
  refl          : {f : BCC a b}
    → f ≈ f
  sym           : {f g : BCC a b}
    → f ≈ g → g ≈ f
  trans         : {f g h : BCC a b}
    → f ≈ g → g ≈ h → f ≈ h
  congl         : {x y : BCC a b} {f : BCC b c}
    → x ≈ y → f • x ≈ f • y
  congr         : {x y : BCC b c} {f : BCC a b}
    → x ≈ y → x • f ≈ y • f
```

Fig. 2. Conversion rules for BCC

Example 4. Given f : BCC a $(b \Rightarrow c)$ and g : BCC a b, if f can be converted to curry f', then the term (apply • pair f g) : BCC a c can be converted to f' • pair id g (the implementation is left as an exercise for the reader). Notice that the combinators curry and apply are eliminated in the result of the conversion. This conversion corresponds to β conversion for functions in STLC, and forms the basis for exponential elimination.

3 OVERVIEW OF NORMALIZATION

Our goal is to implement a normalization algorithm for BCC terms and show that normalization eliminates exponentials. We will achieve the latter using a syntactic property of normal forms called the weak subformula property. To make this property explicit, we define normal forms as a separate data type Nf as follows.

```
data Nf : Ty → Ty → Set where
```

Normal forms are not themselves BCC terms, but they can be embedded into BCC terms using a quotation function q which has the following type.

```
q : Nf a b → BCC a b
```

To prove that normalization eliminates exponentials, we show that normal forms with first-order types can be quoted into a first-order combinator language, called DBC, as follows.

```
qD : firstOrd a → firstOrd b → Nf a b → DBC a b
```

The data type DBC is defined syntactically identical to BCC without the exponential combinators curry and apply, and with an additional distributivity combinator distr (see Section 5).

Normalization based on rewriting techniques performs syntactic transformations of a term to produce a normal form. NbE, on the other hand, normalizes a term by evaluating it in a suitable semantic model, and extracting a normal form from the resulting value. Evaluation is implemented as an interpreter function eval, and extraction of normal forms—also called *reification*—is implemented as a function reify (see Section 6). These functions have the following types.

```
eval : BCC a b → (⟦ a ⟧ ↠ ⟦ b ⟧)
reify : (⟦ a ⟧ ↠ ⟦ b ⟧) → Nf a b
```

The type ⟦ a ⟧ is an interpretation of a BCC type a in the model, and similarly for b. The type ⟦ a ⟧ ↠ ⟦ b ⟧, on the other hand, is a function between interpretations (to be defined later) and denotes the interpretation of a BCC term of type BCC a b.

Normalization is achieved by evaluating a term and then reifiying it, and is thus implemented as a function norm defined as follows.

```
norm : BCC a b → Nf a b
norm t = reify (eval t)
```

To ensure that the normal form generated for a term is correct, we must ensure that it is convertible to the original term. This correctness theorem is stated by quoting the normal form as follows.

correct-nf : $(t : \mathsf{BCC}\ a\ b) \to t \approx \mathsf{q}\ (\mathsf{norm}\ t)$

We prove this theorem using logical relations between BCC terms and values in the semantic model (see Section 7).

4 SELECTIONS

The evaluation of a term requires an input of the appropriate type. During normalization, since we do not have the input, we must assign a reference to the unknown input value and use this reference to represent the value. In lambda calculus, these references are simply variables. Since BCC combinators lack the notion of variables, we must identify the subset of BCC terms which (intuitively) play the counterpart role—which is the goal of this section.

If we think of the typing context as the "input type" of a lambda term, then variables are essentially indices which *project* an unknown value from the input (a substitution). This is because typing contexts enforce a product-like structure on the input. For example, the variable x in the body of lambda term $\Gamma, x : a \vdash x : a$ projects a value of type a from the context $\Gamma, x : a$. The BCC equivalent of $\Gamma, x : a \vdash x : a$ is the term exl : $(\Gamma * a)\ a$. Unlike lambda terms, however, BCC terms do not enforce a specific type structure on the input, and may also return the input entirely as id : $(\Gamma * a)\ (\Gamma * a)$. Hence, as opposed to projections, we need a notion of *selections*.

Specific BCC terms can be used to *select* an unknown value from the input, and these terms can be defined explicitly by the data type Sel (see Figure 3). A term of type Sel $a\ b$ denotes a selection of b from the input a. When the input is a product, the constructor drop drops the second component, and applies a given selection to the first component. The constructor keep, on the other hand, keeps the second component unaltered and applies a selection to the first component. We cannot select further from the input if it is not a product, and hence the remaining constructors, with the prefix end, state that we must simply return the input as is—thereafter referred to as end- constructors.

```
data Sel : Ty → Ty → Set where
  endu : Sel 1 1
  endi  : Sel 0 0
  endb : Sel base base
  ends : Sel (a + b) (a + b)
  ende : Sel (a ⇒ b) (a ⇒ b)
  drop : Sel a b → Sel (a * c) b
  keep : Sel a b → Sel (a * c) (b * c)
```

Fig. 3. Selections

Note that the four end- constructors enable the definition of a unique *identity* selection[2], iden : Sel $a\ a$. This selection can be defined by induction on the type a,

[2]We prefer to derive the identity selection as opposed to adding it as a constructor, to avoid ambiguity which could be created between selections iden and (keep iden), both of the type Sel $(a_1 * a_2)\ (a_1 * a_2)$. The derived identity avoids this ambiguity by definition.

where the only interesting case of products is defined as below. The remaining cases can be defined using the appropriate end- constructor.

```
iden : {a : Ty} → Sel a a
iden {a₁ ∗ a₂} = keep iden
- end- for remaining cases
```

Figure 4 illustrates the use of selections by examples.

$$\begin{aligned}\mathsf{drop}\ \mathsf{iden} &: \mathsf{Sel}\ ((a + b) * c)\ (a + b)\\ \mathsf{keep}\ (\mathsf{drop}\ \mathsf{iden}) &: \mathsf{Sel}\ (a * b * c)\ (a * c)\\ \mathsf{drop}\ (\mathsf{keep}\ (\mathsf{drop}\ \mathsf{iden})) &: \mathsf{Sel}\ (a * b * c * d)\ (a * c)\end{aligned}$$

Fig. 4. Examples of selections

Selections form the basis for the semantic interpretation of BCC terms, and hence enable the implementation of NbE. To this extent, they have the following properties.

Property 4.1 (Category of selections)**.** Selections define a category where the objects are types and a morphism between two types a and b is a selection of type Sel a b. The identity morphisms are defined by iden, and morphism composition can be defined by straight-forward induction on the morphisms as a function of type _∘_ : Sel b c → Sel a b → Sel a c. The identity and associativity laws of a category (sel-idl, sel-idr and sel-assoc below) can be proved using Agda's built-in syntactic equality ≡ by induction on the morphisms. These laws have have the following types in Agda.

```
sel-idl : {s : Sel a b} → iden ∘ s ≡ s
sel-idr : {s : Sel a b} → s ∘ iden ≡ s
sel-assoc : {s₁ : Sel c d} {s₂ : Sel b c} {s₃ : Sel a b}
  → (s₁ ∘ s₂) ∘ s₃ ≡ s₁ ∘ (s₂ ∘ s₃)
```

Property 4.2 (Faithful embedding)**.** Selections can be faithfully embedded into BCC terms since they are simply a subset of BCC terms. This embedding can be implemented by induction on the selection, as follows.

```
embSel : Sel a b → BCC a b
embSel (drop e) = embSel e • exl
embSel (keep e) = pair (embSel e • exl) exr
- id for remaining cases
```

5 NORMAL FORMS

In this section, we present normal forms for BCC terms, and prove exponential elimination using them. It is important to note that these normal forms are *not* normal forms of the conversion rules specified by the relation ≈, but rather are a convenient syntactic restriction over BCC terms for proving exponential elimination. Precisely, they are normal forms of BCC terms which obey a weak subformula property—defined later in this section. This characterization is based on normal forms of proofs in logic, as opposed to normal forms of terms in lambda calculus.

Normal forms are defined mutually with *neutral* forms (see Figure 5). Roughly, neutral forms are eliminators applied to selections, and they represent terms which are blocked during normalization due to unavailability of the input. The neutral form constructor sel embeds a selection as a base case of neutrals; while fst, snd and app represent the composition of the eliminators exl, exr and apply (respectively) to neutrals.

The normal form constructors unit, pair and curry represent their BCC term counterparts; ne-**0** and ne-b embed neutrals which return values of type **0** and base (respectively) into normal forms; left and right represent the composition of the injections inl and inr respectively; and case represents the BCC term case below, which is an eliminator of sums defined using distributivity of products over sums. Note that the BCC term Distr implements this distributivity requirement, and can be derived using exponentials—see Appendix A.2.

```
- Distr : BCC (a * (b + c)) ((a * b) + (a * c))
case' : BCC a (b + c) → BCC (a * b) d → BCC (a * c) d → BCC a d
case' x f g = match f g • Distr • pair id x
```

The quotation functions are implemented as a simple syntax-directed translation by mapping neutrals and normal forms to their BCC counterparts as discussed above. For example, the quotation of the neutral form fst x—where x has the type Ne a $(b * c)$—is simply exl : $(b * c)$ b composed with the quotation of x. Similarly, the quotation of left x is inl composed with the quotation of its argument x. We use the derived term case' to quote the normal form case.

Note that the normal forms resemble $\beta\eta$ long forms of STLC with products and sums [2], but differ with respect to the absence of typing contexts and variables. In place of variables, we use selections in neutral forms—this is an important difference since it allows us to implement *reflection*, a key component of reification (discussed later in Section 6).

In the rest of this section, we will define the weak subformula property, show that all normal forms obey it, and prove exponential elimination as a corollary.

5.1 Weak Subformula Property

To understand the need for a subformula property, let us suppose that we are given a term t : BCC (**1** * **1**) **1**. Does t use exponentials? Unfortunately, we cannot say much about the presence of curry and apply in the subterms without inspecting the body of the term itself. Term t could be something as simple as exl or it could be:

$$\text{apply} \bullet (\text{pair} (\text{curry unit} \bullet \text{exl}) \text{exr}) : \text{BCC} (\mathbf{1} * \mathbf{1}) \mathbf{1}$$

But with an appropriate subformula property, however, this becomes an easy task. Let us suppose that t : BCC (**1** * **1**) **1** has a property that the input and output types of all its subterms occur in t's input (**1** * **1**) and/or output (**1**) type. In this case, what can we say about the presence of curry and/or apply in t? Well, it would not contain any! The input and output types of *all* the subterms would be **1** and/or products of it, and hence it is impossible to find a curry or an apply in a subterm. Let us define this property precisely and show that normal forms obey it by construction.

The occurrence of a type in another is defined as follows.

```
data Nf (a : Ty) : Ty → Set where
  unit   : Nf a 1
  ne-0   : Ne a 0 → Nf a b
  ne-b   : Ne a base → Nf a base
  left   : Nf a b → Nf a (b + c)
  right  : Nf a c → Nf a (b + c)
  pair   : Nf a b → Nf a c → Nf a (b ∗ c)
  curry  : Nf (a ∗ b) c → Nf a (b ⇒ c)
  case   : Ne a (b + c) → Nf (a ∗ b) d → Nf (a ∗ c) d → Nf a d

data Ne (a : Ty) : Ty → Set where
  sel  : Sel a b → Ne a b
  fst  : Ne a (b ∗ c) → Ne a b
  snd  : Ne a (b ∗ c) → Ne a c
  app  : Ne a (b ⇒ c) → Nf a b → Ne a c

q : Nf a b → BCC a b
q unit          = unit
q (ne-b x)      = qNe x
q (ne-0 x)      = init • qNe x
q (left n)      = inl • q n
q (right n)     = inr • q n
q (pair m n)    = pair (q m) (q n)
q (curry n)     = curry (q n)
q (case x m n) = case' (qNe x) (q m) (q n)

qNe : Ne a b → BCC a b
qNe (sel x)     = embSel x
qNe (fst x)     = exl • qNe x
qNe (snd x)     = exr • qNe x
qNe (app x n) = apply • pair (qNe x) (q n)
```

Fig. 5. Normal forms and quotation

Definition 5.1 (Weak subformula)**.** A type b is a weak subformula of a if $b \lhd a$, where $\lhd$ is defined as follows.

```
data _◁_ : Ty → Ty → Set where
  self  : a ◁ a
  subl  : {_⊗_ : BinOp} → a ◁ b → a ◁ (b ⊗ c)
  subr  : {_⊗_ : BinOp} → a ◁ c → a ◁ (b ⊗ c)
  subp  : a ◁ c → b ◁ d → (a ∗ b) ◁ (c ∗ d)
```

For a binary type operator $\otimes$ which ranges over $*$, $+$ or $\Rightarrow$, this definition states that:

- a is a weak subformula of a (self)

- a is a weak subformula of $b \otimes c$ if a is a weak subformula of b (subl) or a is a weak subformula of c (subr)
- $a * b$ is a weak subformula of $c * d$ if a is a weak subformula of c and b is a weak subformula of d (subp).

The constructors self, subl and subr define precisely the concept of a subformula in proof theory [26]. For BCC terms, however, we also need subp which weakens the subformula definition by relaxing it *up to products*. To understand this requirement, we must first define the following property for normal forms.

Definition 5.2 (Weak subformula property)**.** A normal form of type Nf a b obeys the weak subformula property if, for all its subterms of type Nf i o, we have that $i \lhd a * b$ and $o \lhd a * b$.

Do all normal forms obey this property? It is easy to see that the normal forms constructed using unit, left, right and pair obey the weak subformula property given their subterms do the same. For instance, the constructor left returns a normal form of type Nf a $(b + c)$, where the input type (a) and output type (b) of its subterm Nf a b occur in a and $(b + c)$. Hence, if a subterm t : Nf a b obeys the weak subformula property, then so does left t.

To understand why curry satisfies the weak subformula property, recall its definition as a normal form constructor of type BCC $(c * a)$ $b \longrightarrow$ BCC c $(a \Rightarrow b)$. The input type $c * a$ of its subterm argument is evidently not a subformula—as usually defined in proof theory—of the types c or $a \Rightarrow b$. However, by subp, we have that the type $c * a$ is a weak subformula of the *product* of the input and output types $c * (a \Rightarrow b)$. This is precisely the need for weakening the definition of a subformula with subp[3]. Specifically, the proof of $(c * a) \lhd c * (a \Rightarrow b)$ is given by subp (self) (subl self).

On the other hand, the definition of the constructor case looks a bit suspicious since it allows the types b and c which do not occur in final type Nf a d. To understand why case also satisfies the weak subformula property, we must establish the following property about neutral forms, which we shall call *neutrality*.

Property 5.1. Given a neutral form Ne a b, we have that b is a weak subformula of a, i.e., neutrality : Ne a $b \longrightarrow b \lhd a$.

Proof. By induction on neutral forms. For the base case sel, we need a lemma about neutrality of selections, which can be implemented by an auxiliary function neutrality-sel : Sel a $b \longrightarrow b \lhd a$ by induction on the selection. For the other cases, we simply apply the induction hypothesis on the neutral subterm. □

Due to neutrality of the neutral form Ne a $(b + c)$ in the definition of case, we have that $(b + c) \lhd a$, and hence $(b + c) \lhd (a * d)$. As a result, case also obeys the weak subformula property. Similarly, ne-0 and ne-b also obey the weak subformula property as a consequence of neutrality. Thus, we have the following theorem.

Theorem 5.1. All normal forms, as defined by the data type Nf, satisfy the weak subformula property.

[3]In logic, however, the requirement for weakening a subformula by products is absent, since an equivalent definition of curry as $\Gamma, a \vdash b \longrightarrow \Gamma \vdash a \Rightarrow b$ uses context extension (,) instead of products ($*$)

Proof. By induction on normal forms, as discussed above. □

Notice that, unlike normal forms, arbitrary BCC terms need not satisfy the weak subformula property. The term apply • (pair (curry unit • exl) exr) discussed above is already an example of such a term. More specifically, its subterm apply has the input type $(\mathbf{1} \Rightarrow \mathbf{1}) * \mathbf{1}$, which does not occur in $(\mathbf{1} * \mathbf{1}) * \mathbf{1}$—i.e., $(\mathbf{1} \Rightarrow \mathbf{1}) * \mathbf{1} \not\lhd (\mathbf{1} * \mathbf{1}) * \mathbf{1}$. However, all BCC terms, including the ones which do not satisfy the weak subformula property, can be converted to terms which satisfy this property. This conversion is precisely the job of normalization. For instance, the previous example can be converted to unit : BCC $(\mathbf{1} * \mathbf{1})$ $\mathbf{1}$ using uniq-unit. A normalization algorithm performs such conversions automatically whenever possible.

Since neutral forms offer the intuition of an "eliminator", it might be disconcerting to see case, an eliminator of sums, oddly defined as a normal form. But suppose that it was defined in neutrals as follows.

$$\mathsf{case?} : \mathsf{Ne}\ a\ (b + c) \to \mathsf{Nf}\ (a * b)\ d \to \mathsf{Nf}\ (a * c)\ d \to \mathsf{Ne}\ a\ d$$

Such a definition breaks neutrality (Property 5.1) since we cannot prove that $d \lhd a$, and subsequently breaks the weak subformula property of normal forms (Theorem 5.1). But what about the following definition where the first argument to case is normal, instead of neutral?

$$\mathsf{case?} : \mathsf{Nf}\ a\ (b + c) \to \mathsf{Nf}\ (a * b)\ d \to \mathsf{Nf}\ (a * c)\ d \to \mathsf{Nf}\ a\ d$$

Such a definition also breaks the weak subformula property—for the exact same reason which caused our suspicion in the first place: b and c do not occur in a, d or $a * d$.

5.2 Syntactic Elimination of Exponentials

Exponential elimination can be proved as a simple corollary of the weak subformula property of normal forms. If a and b are first-order types, i.e., if the type constructor $\Rightarrow$ does not occur in types a or b, then we can be certain that the subterms of Nf a b do not use curry (from Nf) or app (from Ne). This follows directly from the weak subformula property (Theorem 5.1). To show this explicitly, let us quote such normal forms to a first-order combinator language based on *distributive bicartesian categories* (DBC) [6].

The DBC term language is defined by the data type DBC, which includes all the BCC term constructors except Curry and Apply—although most of them have been left out here for brevity. Additionally, it also has a distributivity constructor distr which distributes products over sums. The constructor distr is needed to implement the BCC term case', which is in turn needed to quote the normal form case (as earlier). This is because distributivity can no longer be derived in the absence of exponentials.

To restrict the input and output to first-order types, suppose that we define a predicate on types, firstOrd : Ty → Set, which disallows the occurrence of exponentials in a type. Given this predicate, we can now define quotation functions qNeD and qD as below. The implementation of the function qNeD is similar to that of the function qNe (discussed earlier) for most cases, and similarly for qD. The only interesting cases are that of the exponentials, and these can be implemented as follows.

```
data DBC : Ty → Ty → Set where
  id    : DBC a a
  _•_ : DBC b c → DBC a b → DBC a c
  -- exl, exr, pair, init
  -- inl, inr, match, unit
  distr : DBC (a * (b + c)) ((a * b) + (a * c))
```

Fig. 6. DBC combinators

```
qNeD : firstOrd a → Ne a b → DBC a b
qNeD p (app n _) = ⊥-elim (expNeutrality p n)

qD : firstOrd a → firstOrd b → Nf a b → DBC a b
qD p q (curry n) = ⊥-elim q
```

For neutrals, we escape having to quote app because such a case is impossible: We have a proof p : firstOrd a which states that input type a does not contain any exponentials. However, the exponential return type of n, say $b \Rightarrow c$, must occur in a by neutrality of n : Ne a $(b \Rightarrow c)$—which contradicts the proof p. Hence, such a case is not possible. This reasoning is implemented by applying the function ⊥-elim with a proof of impossibility produced using an auxiliary function expNeutrality : firstOrd $a \to$ Ne $a\ b \to$ firstOrd b. Similarly, we escape the quotation of the normal form curry since Agda automatically inferred that such a case is impossible. This is because a proof q which states that the output b is not an exponential, is contradicted by the definition of curry which states that it must be—hence q must be impossible.

Although we have shown the syntactic elimination of exponentials using normal forms, we are yet to show that there exists an equivalent normal form for every term. For this, we must implement normalization and prove its correctness.

6 NORMALIZATION FOR BCC

To implement evaluation and reification, we must first define an appropriate interpretation for types and terms. A naive Set-based interpretation (such as ⟦_⟧n below) which maps BCC types to their Agda counterparts fails quickly.

```
⟦ 1       ⟧n = ⊤
⟦ 0       ⟧n = ⊥
⟦ base    ⟧n = ??
⟦ t₁ * t₂ ⟧n = ⟦ t₁ ⟧n × ⟦ t₂ ⟧n
⟦ t₁ + t₂ ⟧n = ⟦ t₁ ⟧n ⊎ ⟦ t₂ ⟧n
⟦ t₁ ⇒ t₂ ⟧n = ⟦ t₁ ⟧n → ⟦ t₂ ⟧n
```

What should be the correct interpretation of the type base? The naive interpretation also makes it impossible to implement reflection for the empty and sum types, since their inhabitants cannot be faithfully represented in such an interpretation (see Section 6.3). To address this problem, we must first define an appropriate semantic model.

6.1 Interpretation in Presheaves

To implement NbE, our choice of semantic model for interpretation of BCC types must allow us to implement both evaluation and reification. NbE for STLC can be implemented by interpreting it in *presheaves* over the category of *weakenings* [4] [2]. The semantic equivalence of BCC combinators and STLC suggests that it should be possible to interpret BCC terms in presheaves as well. The difference, however, is that we will interpret BCC in presheaves over the category of selections (instead of weakenings). Such a presheaf, for our purposes, is simply the following record definition:

```
record Pre : Set₁ where
  field
    In : Ty → Set
    lift : {i j : Ty} → Sel j i → (In i → In j)
```

Intuitively, an occurrence In i can be understood as a Set interpretation indexed by an input type i. The function lift can be understood as a utility function which converts a semantic value for the input i to a value for a "larger" input j, for a given selection of i from j.

For the category theory-aware reader, notice that Pre matches the expected definition of a presheaf as a functor which maps objects (using In) and morphisms (using lift) in the opposite category of the category of selections to the Set-category. We skip the functor laws of the presheaf in the Pre record to avoid cluttering the normalization procedure, and instead prove them separately as needed for the correctness proofs later.

With the definition of a presheaf, we can now implement the desired interpretation of types as ⟦_⟧ : Ty → Pre. Intuitively, a presheaf model allows us to interpret a BCC type as an Agda type *for* a given input type—or equivalently for a given typing context. To implement the function ⟦_⟧, we will need various presheaf constructions (instances of Pre)—defining these is the goal of this section. Note that all names ending with ' denote a presheaf.

```
1' : Pre                    0' : Pre
1' .In _     = ⊤            0' .In _ = ⊥
1' .lift _ _ = tt           0' .lift _ ()
```

Fig. 7. Unit and Empty presheaves

The unit presheaf maps all input types to the type ⊤ (unit type in Agda) and empty presheaf maps it to ⊥ (empty type in Agda) (see Figure 7). The implementation of lift is trivial in both cases since the only inhabitant of ⊤ is tt, and ⊥ has no inhabitants.

The product of two presheaves A and B is defined component-wise as follows.

```
_*'_ : Pre → Pre → Pre
(A *' B) .In i          = A .In i × B .In i
(A *' B) .lift s (x , y) = (A .lift s x , B .lift s y)
```

The function lift is implemented component-wise since s has the type Sel j i, x has the type A .In i, y has the type B .In i, and the result must be a value of type A .In $j \times B$ .In j. Similarly, the sum of two presheaves is also defined component-wise as follows.

```
_+'_ : Pre → Pre → Pre
(A +' B) .In i = A .In i ⊎ B .In i
(A +' B) .lift s (inj₁ x) = inj₁ (A .lift s x)
(A +' B) .lift s (inj₂ x) = inj₂ (B .lift s x)
```

It is tempting to implement an exponential presheaf _⇒'_ component wise (like _x'_), but this fails at the implementation of lift: given Sel j i, we can not lift a function (A .In $i \to B$ .In i) to (A .In $j \to B$ .In j) directly. To solve this, we must implement a slightly more general version which allows for lifting as follows.

```
_⇒'_ : Pre → Pre → Pre
(A ⇒' B) .In i = {i₁ : Ty} → Sel i₁ i → A .In i₁ → B .In i₁
(A ⇒' B) .lift s f s' = f (s ∘ s')
```

Recall that the operator ∘ implements composition of selections. The interpretation of the exponential presheaf is defined for a given input type i, as a function (space) for all selections of the type i_1 from i [18]—which gives us the required lifting by composition of the selections.

BCC terms also define presheaves when indexed by the output type.

```
BCC' : Ty → Pre
BCC' o .In i = BCC i o
BCC' o .lift s t = liftBCC s t
```

To implement liftBCC, recollect that selections can be embedded into BCC terms using the embSel function (from Section 4). Hence, lifting BCC terms can be implemented easily using composition, as follows.

```
liftBCC : Sel j i → BCC i a → BCC j a
liftBCC s t = t • embSel s
```

Similarly, normal forms and neutral forms also define presheaves when indexed by the output type (see Figure 8). The implementation of lift for normal forms (litfNf) can be defined by straight-forward induction on the normal form—and similarly for liftNe.

```
Nf' : Ty → Pre                Ne' : Ty → Pre
Nf' o .In i = Nf i o          Ne' o .In i = Ne i o
Nf' o .lift s n = liftNf s n  Ne' o .lift s n = liftNe s n
```

Fig. 8. Normal and Neutral form presheaves

For notational convenience, let us define a type alias Sem for values in the interpretation:

```
Sem : Ty → Pre → Set
Sem x P = P .In x
```

For example, a value of type Sem a ⟦ b ⟧ denotes a "semantic value" in the interpretation ⟦ b ⟧ indexed by the input type a. When the input is irrelevant, we simply skip mentioning it and say "value in the interpretation".

A BCC term is interpreted as a *natural transformation* between presheaves, which is defined as follows.

```
_↠_ : Pre → Pre → Set
A ↠ B = {i : Ty} → Sem i A → Sem i B
```

Intuitively, this function maps semantic values in A to semantic values in B (for the same input type i).

6.2 NbE for CCC Fragment

NbE for the fragment of BCC which excludes the empty and sum types, namely the CCC fragment, is rather simple—let us implement this first in this section. The presheaves defined in the previous section allow us to address the issue from earlier for interpreting the type base. The interpretation for types in the CCC fragment is defined as follows.

```
⟦_⟧ : Ty → Pre
⟦ 1 ⟧       = 1'
⟦ base ⟧    = Nf' base
⟦ a ∗ b ⟧   = ⟦ a ⟧ ∗' ⟦ b ⟧
⟦ a ⇒ b ⟧   = ⟦ a ⟧ ⇒' ⟦ b ⟧
```

The unit, product and exponential types are simply interpreted as their presheaf counterparts. The base type, on the other hand, is interpreted as the presheaf of normal forms indexed by base. This is because the definition of BCC has no combinators specifically for base types, which means that a term BCC i base must depend on its input for producing a base value. Hence, we interpret it as a family of normal forms which return base for any input i—which is precisely the definition of the Nf' base presheaf. Note that this interpretation of base types is fairly standard [17].

Having defined the interpretation of types, we can now define the interpretation of BCC terms, i.e., evaluation, as follows.

```
eval : BCC a b → (⟦ a ⟧ ↠ ⟦ b ⟧)
eval id       x       = x
eval (f • g)  x       = eval f (eval g x)
eval unit     x       = tt
eval exl      (x₁ , _) = x₁
eval exr      (_ , x₂) = x₂
eval (pair t₁ t₂) x   = eval t₁ x , eval t₂ x
eval apply    (f , x) = f iden x
eval {a} (curry t) x = λ s y → eval t (lift ⟦ a ⟧ s x , y)
```

The function eval interprets the term id as the the identity function, term composition • as function composition, exl as the first projection, and so on for the other simple cases. Let us take a closer look at the exponential fragment.

To interpret apply for a given function f (of type Sem i ⟦ $a_1 \Rightarrow a_2$ ⟧) and its argument x (of type Sem i ⟦ a_1 ⟧), we must return a value for its application (of type Sem i ⟦ a_2 ⟧). Recollect from the definition of the exponential presheaf that an exponential is interpreted as a generalized function for a given selection. In this case, we do not need this generality since the function and its argument are both semantic values for the same input type i. Hence, we simply use the identity selection iden : Sel i i, to obtain a suitable function which accepts the argument y .

The interpretation of a term curry t (of type BCC a ($b_1 \Rightarrow b_2$)) for a given x (of type Sem i ⟦ a ⟧) must be a function (of type Sem i_1 ⟦ $b_1 \Rightarrow b_2$ ⟧) for a given selection s (of type Sel i_1 i). We achieve this by recursively evaluating t (of type BCC ($a * b_1$) b_2), with a pair of arguments (of type Sem i_1 ⟦ a ⟧ and Sem i_1 ⟦ b_1 ⟧). For the first component, we could use x, but since it is a semantic value for the input i instead of i_1, we must first lift it to i_1 using the selection s—which explains the occurrence of lift.

To implement the reification function reify : (⟦ a ⟧ ↠ ⟦ b ⟧) ⟶ Nf a b, we need two natural transformations: reflect : Ne' a ↠ ⟦ a ⟧ and reifyVal : ⟦ b ⟧ ↠ Nf' b. The former converts a neutral to a semantic value, and the latter extracts a normal form from the semantic value. Using these functions, we can implement reification as follows.

```
reify : (⟦ a ⟧ ↠ ⟦ b ⟧) ⟶ Nf a b
reify {a} f = let y = reflect {a} (sel iden)
              in reifyVal ( f y )
```

The main idea here is the use of reflection to produce a value of type Sem a a. This value enables us to apply the function f to produce a result of type Sem a ⟦ b ⟧. The resulting value is then used to apply reifyVal and return a normal form of type Nf a b.

The natural transformations used in reification are implemented as follows.

```
reflect : {a : Ty} ⟶ Ne' a ↠ ⟦ a ⟧
reflect {1}          x = tt
reflect {base}       x = ne-b x
reflect {a₁ * a₂}    x = reflect {a₁} (fst x) , reflect {a₂} (snd x)
reflect {a₁ ⇒ a₂} x = λ s y ⟶
  reflect {a₂} (app (liftNe s x) (reifyVal y))

reifyVal : {b : Ty} ⟶ ⟦ b ⟧ ↠ Nf' b
reifyVal {1}          x = unit
reifyVal {base}       x = x
reifyVal {b₁ * b₂}    x = pair (reifyVal (proj₁ x)) (reifyVal (proj₂ x))
reifyVal {b₁ ⇒ b₂} x =
  curry (reifyVal (x (drop iden) (reflect {b₁} (snd (sel iden)))))
```

Reflection is implemented by performing a type-directed translation of neutral forms to semantic values. For example, in the product case, a pair is constructed by recursively reflecting the components of the neutral. For the exponential case, the reflection of a neutral x must return a function which accepts a selection s, an argument y, and returns a semantic value for the application of the neutral x with the argument y. In other words, the body of the function needs to be constructed somehow by applying x (a neutral function) with argument y (a semantic value). The neutral application constructor app has two requirements: the function and the argument must accept the same input, and the argument must be in normal form. To satisfy the first requirement, we lift the neutral x using the selection s, and for the latter requirement we reify the argument value y. Finally, we reflect the neutral application to produce the desired semantic value.

The implementation of the function reifyVal is similar to reflection, but performs the dual action: producing a normal form from a semantic value. Like reification, we implement this by type-directed translation of semantic values to normal forms. Notice that the case of base type is trivial for both functions. This is because we defined the interpretation of base types as normal forms (Nf' base), and a semantic value is already in normal form. Hence, reifyVal simply returns the semantic value, and reflection applies ne-b on the neutral to construct a normal form.

6.3 NbE for Sums and Empty Type

Let us suppose that we interpret **0** as ⟦ **0** ⟧ = **0'**. Now consider extending the implementation of reflection for the following case:

```
reflect {0} y = ??
```

How should we handle this case? The types tell us that we need to construct a semantic value of the type ⊥ (recollect the definition of **0'**). Since ⊥ is an empty type, this is an impossible task! A similar problem arises for sums when we interpret them as ⟦ $a + b$ ⟧ = ⟦ a ⟧ +' ⟦ b ⟧. Reflection requires us to make a choice over a returning a semantic value of ⟦ a_1 ⟧ or ⟦ a_2 ⟧. Which is the right choice? Unfortunately, we cannot make a decision with the given information since it could be either of the cases.

We cannot construct the impossible or decide over the component of a sum to reflect, hence we will simply build up a tree of decisions that we do not wish to make. A decision tree is defined inductively by the following data type:

```
data Tree (i : Ty) (P : Pre) : Set where
  leaf   : Sem i P → Tree i P
  dead   : Ne i 0 → Tree i P
  branch : Ne i (a + b) → Tree (i ∗ a) P → Tree (i ∗ b) P → Tree i P
```

A leaf in a decision tree can be leaf, in which case it contains a semantic value in P. Alternatively, a leaf can also be dead, in which case it contains a neutral which returns **0**. A branch of the tree is constructed by branch, and represents the choice over a neutral form which returns a coproduct.

Intuitively, a tree represents a suspended computation for a value in the interpretation P. For example, Tree i **0'** represents a suspended computation for a value in Sem

i **0**'—which is ⊥. Since values of this type are impossible, all the leaves of such a tree must be dead. Similarly, a tree Tree *i* ⟦ *a* + *b* ⟧ represents a suspended computation for a value of type Sem *i* ⟦ *a* + *b* ⟧—which is a sum of Sem *i* ⟦ *a* ⟧ and Sem *i* ⟦ *b* ⟧.

Trees define a monad Tree' on presheaves as follows.

```
Tree' : Pre → Pre
(Tree' A) .In i = Tree i A
(Tree' a) .lift = liftTree
```

The function liftTree is defined by induction on the tree. The standard monadic operations return, map and join are defined by the following natural transformations:

```
return : ∀ {P} → P ↠ Tree' P
join : ∀ {P} → Tree' (Tree' P) ↠ Tree' P
map : ∀ {P Q} → (P ↠ Q) → Tree' P ↠ Tree' Q
```

The natural transformation return is defined as leaf, while join and map can be defined by straight-forward induction on the tree. The monadic structure of trees are precisely the reason they allow us to represent suspended computation.

With the tree monad, we can now complete the interpretation of types **0** and + as follows.

```
⟦ 0 ⟧     = Tree' 0'
⟦ a + b ⟧ = Tree' (⟦ a ⟧ +' ⟦ b ⟧)
```

By interpreting the empty and sum types in the Tree' monad, we are able to handle the problematic cases of reflection by returning a value in the monad, as follows.

```
reflect {0}     x = dead x
reflect {a + b} x = branch x
                      (leaf (inj₁ (reflect {a} (snd (sel iden)))))
                      (leaf (inj₂ (reflect {b} (snd (sel iden)))))
```

In addition to general monadic operations, the monad Tree' also supports the following special "run" operations:

```
runTree : Tree' ⟦ a ⟧ ↠ ⟦ a ⟧
runTreeNf : Tree' (Nf' a) ↠ Nf' a
```

These natural transformations allow us to run a monadic value to produce a regular semantic value, and are required to implement eval and reifyVal. The implementation of these natural transformations is mostly mechanical: runTreeNf can be defined by induction on the tree, and runTree can be defined by induction on the type *a* using an "applicative functor" map Tree *c* ⟦ *a* ⇒ *b* ⟧ → Tree c ⟦ *a* ⟧ → Tree c ⟦ *b* ⟧ for the exponential case.

The remaining cases of evaluation are implemented as follows.

```
eval             inl          x = return (inj₁ x)
eval             inr          x = return (inj₂ x)
eval {0}     {b} init         x = runTree {b} (map cast x)
eval {a + b} {c} (match f g) x = runTree {c} (map match' x)
  where
```

```
  match' : (⟦ a ⟧ +' ⟦ b ⟧) ↠ ⟦ c ⟧
  match' (inj₁ y) = eval f y
  match' (inj₂ y) = eval g y
```

For the case of inl, we have a semantic value x in the interpretation $[\![a]\!]$, and we need a monadic value Tree' ($[\![a]\!]$ +' $[\![b]\!]$). To achieve this, we simply return the value in the monad by applying the injection inj_1. The case of inr is very similar.

For the case of init, we have a value x in the interpretation Tree' **0'**, and we need a value in $[\![b]\!]$. Since x is a tree, we can map over it using a function cast : **0'** ↠ $[\![b]\!]$ to get a value in Tree' $[\![b]\!]$. The resulting tree can then be run using runTree to return the desired result in $[\![b]\!]$. The function cast has a trivial implementation with an empty body since a value in the interpretation by **0'** has type ⊥. The implementation of match is also similar, and we use a natural transformation match' instead of cast to map over x.

The implementation of reification for the remaining fragment resembles evaluation:

```
reifyVal {0}     x = runTreeNf (map cast x)
reifyVal {a + b} x = runTreeNf (map matchNf x)
  where
    matchNf : (⟦ a ⟧ +' ⟦ b ⟧) ↠ Nf' (a + b)
    matchNf (inj₁ y) = left (reifyVal y)
    matchNf (inj₂ y) = right (reifyVal y)
```

We use the natural transformation runTreeNf instead of runTree and matchNf instead of match.

7 CORRECTNESS OF NORMAL FORMS

A normal form is *correct* if it is convertible to the original term when quoted. The construction of the proof for this theorem is strikingly similar to the implementation of normalization. Although the details of the proof are equally interesting, we will only discuss the required definitions and sketch the proof of the main theorems to keep this section concise. We encourage the curious reader to see the implementation of the full proof for further details (see A.1 for link). We will prove the correctness of normalization by showing that evaluation and reification are correct. To enable the definition of correctness for these functions, we must first relate terms and semantic values using logical relations.

7.1 Kripke Logical Relations

We will prove the correctness theorem using Kripke logical relations à la Coquand [11]. In this section, we define these logical relations.

Definition 7.1 (Logical relation R)**.** A relation R between terms and semantic values, indexed by a type b, is defined by induction on b:

```
R : {b a : Ty} → BCC a b → Sem a ⟦ b ⟧ → Set
R {1}     t v = ⊤
R {base}  t v = t ≈ q v
```

```
R {b₁ * b₂}   t v = R (exl • t) (proj₁ v) × R (exr • t) (proj₂ v)
R {b₁ ⇒ b₂} t v = ∀ {c t' x} → (s : Sel c _)
  → R t' x → R (apply • pair (liftBCC s t) t') (v s x)
R {𝟘}         t v = R₀ t v
R {b + c}     t v = R₊ t v
```

Intuitively, the relation R establishes a notion of equivalence between terms and semantic values, but we will say *related* instead of equivalent to be pedantic. For example, for the case of products, it states that composing the combinator exl with a term is related to applying the projection proj_1 on a value—and similarly for exr and proj_2. In the unit case, it states that terms and values are trivially related. For base types, it states that terms must be convertible to the quotation of values, since values are normal forms by definition of ⟦_⟧. For the case of exponentials, the definition states that t, which returns an exponential, is related to a functional value v, if for all related "arguments" t' and x, the resulting values of the application are related. As usual, since v is a function generalized over selections, the relation also states that it must hold for all appropriate selections.

For the case of empty and sum types, we need a relation between terms and trees—which is defined by Rt as follows.

Definition 7.2 (Logical relation Rt)**.** A relation Rt between terms and trees, indexed by another relation *Rl* between terms and values in the leaves, is defined by induction on the tree:

```
Rt : {B' : Pre} → (Rl : ∀ {a₁} → BCC a₁ b → Sem a₁ B' → Set)
  → BCC a b → Tree a B' → Set
Rt Rl t (leaf a)   = Rl t a
Rt Rl t (dead x) = t ≈ init • qNe x
Rt Rl t (branch x v₁ v₂) = ∃₂ λ t₁ t₂
  → (Rt Rl t₁ v₁) × (Rt Rl t₂ v₂) × (t ≈ case' (qNe x) t₁ t₂)
```

Intuitively, the relation Rt states that a term is related to a tree if the term is related to the values in the leaves. The key idea in the definition of Rt for the leaf case is to parameterize the definition by a relation *Rl* between terms and leaf values. Note that the relation R cannot be used here (instead of a parameterized relation *Rl*) since its type is more specific than the relation needed for leaves. For the case of dead leaves with a neutral returning 𝟘, the definition states that the t must be convertible to elimination of 𝟘 using init. In the branch case, it states the inductive step: t is related to a decision branch in the tree, if t is convertible to a decision over the neutral x (implemented by case') for some t_1 and t_2 related to subtrees v_1 and v_2.

Using the relation Rt, we can now define the remaining relations for the empty and sum types as follows.

Definition 7.3 (Logical relations R_0 and R_+)**.** Logical relations R_0 and R_+ are defined as special cases of Rt using the below defined relations Rl_0 and Rl_+ respectively:

```
Rl₀ : BCC a 𝟘 → Sem a 𝟘' → Set
Rl₀ _ ()
```

```
R₀ : BCC a 𝟎 → Tree a 𝟎' → Set
R₀ t v = Rt Rl₀ t v

Rl₊ : BCC a (b + c) → Sem a (⟦ b ⟧ +' ⟦ c ⟧) → Set
Rl₊ t (inj₁ x) = ∃ λ t' → R t' x × (inl • t' ≈ t)
Rl₊ t (inj₂ y) = ∃ λ t' → R t' y × (inr • t' ≈ t)

R₊ : BCC a (b + c) → Tree a (⟦ b ⟧ +' ⟦ c ⟧) → Set
R₊ t c = Rt Rl₊ t c
```

The relation Rl_0 is simply a type cast since a value of type $\mathsf{Sem}\ a\ \mathbf{0}'$ does not exist. On the other hand, the relation Rl_+, states that t is related to an injection $\mathsf{inj}_1\ x$, if t is convertible to $\mathsf{inl} \bullet t'$ for some t' related to x—and similarly for inj_2 and inr.

7.2 Proof of Correctness

We prove the main correctness theorem (Theorem 7.3) using two intermediate theorems, namely the *fundamental theorem* of logical relations (Theorem 7.1) and the correctness of reification (Theorem 7.2), and various lemmata. In all the cases, we either perform induction on the return type of a term or on a tree. The main idea here is that the appropriate induction triggers the definition of the relations, hence enabling Agda to refine the proof goal for a specific case.

Lemma 7.1 (Invariance under conversion)**.** If a term t is convertible to t' and t' is related to a semantic value v, then t is related to v.

```
invariance : {t t' : BCC a b} {v : Sem a ⟦ b ⟧} → t ≈ t' → R t' v → R t v
```

Proof. By induction on the return type of t and t'. The proof is fairly straightforward equational reasoning using the conversion rules ($\approx$). The empty and sum types can be handled by induction on the tree. □

Lemma 7.2 (Lifting preserves relations)**.** If a term $t : \mathsf{BCC}\ a\ b$ is related to a value $v : \mathsf{Sem}\ a\ ⟦\ b\ ⟧$, then lifting the term is related to lifting the value, for any applicable selection s.

```
liftPresR : {s : Sel c a} {t : BCC a b} {v : Sem a ⟦ b ⟧}
          → R t v → R (liftBCC s t) (lift ⟦ b ⟧ s v)
```

Proof. By induction on the return type of t. As in the previous lemma, the empty and sum types can be handled by induction on the tree. □

Definition 7.4 (Fundamental theorem)**.** If a term t' is related to a semantic value v, then the composition $t \bullet t'$ is related to the evaluation of t with the input v, for all terms t. That is, the fundamental theorem holds if $\mathsf{Fund}\ t$ (defined below) holds for all t.

```
Fund : (t : BCC a b) → Set
Fund {a} {b} t = {c : Ty} {t' : BCC c a} {v : Sem c ⟦ a ⟧}
      → R t' v → R (t • t') (eval t v)
```

Theorem 7.1 (Correctness of evaluation)**.** The fundamental theorem holds, or equivalently, evaluation is correct.

```
correctEval : (t : BCC a b) → Fund t
```

Proof. By induction on the term t. Most cases are proved by the induction hypothesis and some equational reasoning . To enable equational reasoning, we must use the invariance lemma (Lemma 7.1). For the case of curry, the key step is to make use of the β rule for functions (from Section 2).

For the sum and empty types, recall that evaluation uses the natural transformation `runTree : Tree' ⟦ a ⟧ ↠ ⟦ a ⟧`. Hence, to prove correctness of evaluation for these cases, we need a lemma `correctRunTree : Rt R t v → R t v`—which can be proved by induction on the return type of t. The proof of this lemma also requires us to prove correctness of all the natural transformations used by runTree, which can be achieved in similar fashion to correctRunTree. Note that we must use the lifting preservation lemma (Lemma 7.2), wherever lifting is involved, for example, in the curry case. □

Lemma 7.3 (Correctness of reflect and reifyVal)**.** i) The quotation of a neutral form n is related to its reflection. ii) If a term t is related to a value v, then t must be convertible to the normal form which results from the quotation of reification of v.

```
correctReflect : {n : Ne a b} → R (qNe n) (reflect n)
correctReifyVal : {t : BCC a b} {v : Sem a ⟦ b ⟧} → R t v → t ≈ q (reifyVal v)
```

Proof. Implemented mutually by induction on the return type of the neutral / term and using the invariance lemma (Lemma 7.1) to do equational reasoning. Appropriate eta conversion rules are needed for products, exponentials and sums. □

Theorem 7.2 (Correctness of reification)**.** The fundamental theorem proves that t is convertible to quotation of the value obtained by evaluating and reifying t.

```
correctReify : {t : BCC a b} → (Fund t) → t ≈ q (reify (eval t))
```

Proof. By induction on the return type of term t. This theorem follows from Lemma 7.3 and the other lemmata discussed above. □

Theorem 7.3 (Correctness of normal forms). A term is convertible to the quotation of its normal form.

Proof. Since normalization is defined as the composition of reification and evaluation, the correctness of normal forms follows from the correctness of reification and evaluation:

```
correctNf : (t : BCC a b) → t ≈ q (norm t)
correctNf t = correctReify (correctEval t)
```

□

7.3 Exponential Elimination Theorem

Using the syntactic elimination of exponentials illustrated earlier using normal forms (Section 5.2), and the normalization procedure which converts BCC terms to normal forms (Section 6), we finally have the following exponential elimination theorem for BCC terms.

Theorem 7.4 (Exponential elimination)**.** Given that a and b are first-order types, every term $f : \mathsf{BCC}\ a\ b$ can be converted to an equivalent term $f' : \mathsf{DBC}\ a\ b$ which does not use any exponentials.

Proof. From the normalization function norm implemented in Section 6, and the correctness of normal forms by Theorem 7.3, we know that there exists a normal form $n : \mathsf{Nf}\ a\ b$ resulting from the application $\mathsf{norm}\ f$ such that $f \approx \mathsf{q}\ n$. Since a and b are first-order types, we also have a DBC term $\mathsf{qD}\ n : \mathsf{DBC}\ a\ b$, which does not use exponentials by construction. Additionally, since the function qD is a restriction map of the function q, $\mathsf{qD}\ n$ must be equivalent to $\mathsf{q}\ n$, and hence to f. This can be shown by proving that the embedding of the DBC term $\mathsf{qD}\ n$ into BCC is convertible to $\mathsf{q}\ n$, and hence to f. Thus we have an equivalent DBC term $f' = \mathsf{qD}\ n$. □

8 SIMPLICITY, AN APPLICATION

Simplicity is a typed combinator language for programming smart contracts in blockchain applications [24]. It was designed as an alternative to Bitcoin Script, especially to enable static analysis and estimation of execution costs. The original design of Simplicity only allows unit, product and sum types. It does not allow exponentials, the empty type or base types. The simple nature of these types enables calculation of upper bounds on the time and memory requirements of executing a Simplicity program in an appropriate execution model. For example, the bit-size of a value is computed using its type as follows.

$$
\begin{aligned}
\mathsf{size}\ \mathbf{1} &= 0 \\
\mathsf{size}\ (t_1 * t_2) &= \mathsf{size}\ t_2\ \text{`+}\ \mathsf{size}\ t_2 \\
\mathsf{size}\ (t_1 + t_2) &= 1\ \text{`+}\ \mathsf{max}\ (\mathsf{size}\ t_1)\ (\mathsf{size}\ t_2)
\end{aligned}
$$

Note that the operator `+ is simply addition for natural numbers renamed to avoid name clash with the constructor +. The additional bit is need in the sum case to represent the appropriate injection.

Despite Simplicity's ability to express any finite computation between the allowed types, its low-level nature makes it cumbersome to actually write programs since it lacks common programming abstractions such as functions and loops. Even as a compilation target, Simplicity is too low-level. For example, compiling functional programs to Simplicity burdens the compiler with the task of defunctionalization since Simplicity does not have a corresponding notion of functions. To solve this issue, Valliappan et al. [27] note that Simplicity can be modeled in (distributive) bicartesian categories, and propose extending Simplicity with exponentials, and hence to bicartesian closed categories without the empty type.

Although extending Simplicity with exponentials makes it more expressive, it complicates matters for static analysis. For example, the extension of the size function is already a matter of concern:

```
size (t₁ ⇒ t₂) = size t₂ ^ size t₁
```

Valliappan et al. [27] avoid this problem by extending the bit machine with the ability to implement closures, but the problem of computing an upper bound on execution time and memory consumption remains open. Exponential elimination provides a solution for this: Simplicity programs with exponentials can be compiled by eliminating exponentials to programs without exponentials, hence providing a more expressive higher-order target language—while also retaining the original properties of static analysis.

Since Simplicity resembles BCC and DBC combinators, they can be translated to BCC, and from DBC in a straight-forward manner [27]:

```
SimplToBCC : Simpl a b → BCC a b
DBCToSimpl : DBC a b → Simpl a b
```

Exponential elimination bridges the gap between BCC and DBC terms:

```
elimExp : firstOrd a → firstOrd b → BCC a b → DBC a b
elimExp p q t = qD p q (norm t)
```

Thus, we can implement an exponential elimination algorithm for Simplicity programs:

```
elimExpS : firstOrd a → firstOrd b → Simpl a b → Simpl a b
elimExpS p q t = DBCToSimpl (elimExp p q (SimplToBCC t))
```

The difference between the input and output programs is of course that the input may have exponentials, but the output *will not*. The requirements that the input and output of the entire program be first-order types is a harmless one since such programs must have an observable input and output anyway.

Note that we have overlooked the empty type and the combinator `init` in the translation of `DBCToSimpl` here. However, this can be mitigated easily by adding an additional predicate `nonEmpty : Ty → Set` to discharge this case —as in Section 5.2, thanks to the weak subformula property!

Although our work shows that it is possible to eliminate exponentials from Simplicity programs, the implementation provided here might not be the most practical one. Normal forms are in η-expanded form, which means that the generated programs may be much larger than necessary, hence leading to code explosion. Moreover, the translation to BCC and from DBC is also an unnecessary overhead. It may be possible to tame code explosion by normalizing without η expansion [17]. The latter problem, on the other hand, can be solved easily by implementing exponential elimination directly on Simplicity programs. We leave these improvements as suggestions for future work.

9 RELATED WORK

Selections resemble *weakenings* (also called *order preserving embeddings*) in lambda calculus [4]. Weakenings are defined for typing contexts such that a weakening $\Gamma \sqsubseteq \Delta$ selects a "subcontext" Δ from the context Γ [19]. Selections, on the other hand, are simply a subset of BCC terms that select components of the input. Conceptually, selections are the `BCC`-equivalent of weakenings and they have properties (discussed

in Section 4) similar to weakenings. Most importantly, selections unify the notion of weakenings and variables—since they are used in neutrals (as "variables") and for lifting (as "weakenings").

Altenkirch et al. [3] implement NbE to solve the decision problem for STLC with all simple types except the empty type ($\lambda_{\Rightarrow 1 * +}$). Balat et al. [7] solve the *extensional* normalization problem using NbE for the STLC including the empty type ($\lambda_{\Rightarrow 1 * + 0}$). Abel and Sattler [2] provide an account of NbE for $\lambda_{\Rightarrow 1 * + 0}$ using decision trees—the techniques of which they go on to use for more advanced calculi. They in turn attribute the idea of decision trees for normalizing sum types to Altenkirch and Uustalu [5]. Our interpretation model is based on that of Abel and Sattler [2] and the generated normal forms are not unique—caused by commuting case conversions and the overlap between selections and projections. The primary difference between earlier efforts and our work is that we implement NbE for a combinator language.

Altenkirch and Uustalu [5] also prove correctness of normal forms using logical relations, but only for closed lambda terms. Our logical relations have a much more general applicability since they are indexed by the input (or equivalently by the typing context). Moreover, we prove correctness for interpreting sums using decision trees by the means of logical relations generalized over arbitrary presheaf interpretations. Since the decision tree monad Tree' is a *strong monad* [21], it should be possible to further extend this proof technique to normalization of calculi with *computational effects* [15] [2].

10 FINAL REMARKS

We have shown that BCC terms of first-order types can be normalized to terms in a sub-language without exponentials based on distributive bicartesian categories. To this extent, we have implemented normalization using normalization by evaluation, and shown that normal forms are convertible to the original term in the equational theory of BCCs. Moreover, we have also shown the applicability of our technique to erase exponentials from a combinator language called Simplicity. Our work enables a closure-free implementation of BCC combinators and answers previously open questions about the elimination of exponentials.

As noted earlier, the normal forms of BCC combinators presented here are not normal forms of the equational theory specified by the conversion relation $\approx$. This is because the syntax of normal forms does not enforce normal forms of equivalent terms to be unique. For example, the normal forms ne-b (sel (drop endb)) and ne-b (fst (sel (keep endb))) are syntactically different, but inter-convertible when quoted. Hence, the normalization procedure does not derive the conversion relation $\approx$, and cannot be used to decide it. Instead, our notion of normal forms is characterized by the weak subformula property, and aimed at the eliminating intermediate values by restricting the unruly composition which allows introduction and elimination of arbitrary values.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments on earlier drafts of this paper. We thank Andreas Abel for suggesting NbE and for the many discussions about implementing and proving the correctness of NbE. The first author took inspiration

for this work from Andreas' lecture notes on NbE for intuitionistic propositional logic at the Initial Types Club. We would also like to thank Thierry Coquand, Fabian Ruch and Sandro Stucki for the insightful discussions on the topic of exponential elimination. This work was funded by the Swedish Foundation for Strategic Research (SSF) under the projects Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011), as well as the Swedish research agency Vetenskapsrådet.

REFERENCES

[1] Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. 1991. Explicit substitutions. *Journal of functional programming* 1, 4 (1991), 375–416.

[2] Andreas Abel and Christian Sattler. 2019. Normalization by Evaluation for Call-by-Push-Value and Polarized Lambda-Calculus. *arXiv preprint arXiv:1902.06097* (2019).

[3] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Phil Scott. 2001. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 303–310.

[4] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical reconstruction of a reduction free normalization proof. In *International Conference on Category Theory and Computer Science*. Springer, 182–199.

[5] Thorsten Altenkirch and Tarmo Uustalu. 2004. Normalization by evaluation for $\lambda\!\!\rightarrow 2$. In *International Symposium on Functional and Logic Programming*. Springer, 260–275.

[6] Steve Awodey. 2010. *Category theory*. Oxford University Press.

[7] Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*, Vol. 4. 49.

[8] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalization by evaluation. In *Prospects for Hardware Foundations*. Springer, 117–137.

[9] Ulrich Berger and Helmut Schwichtenberg. 1991. An inverse of the evaluation functional for typed lambda-calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 203–211.

[10] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

[11] Catarina Coquand. 1993. From semantics to rules: A machine assisted analysis. In *International Workshop on Computer Science Logic*. Springer, 91–105.

[12] Guy Cousineau, P-L Curien, and Michel Mauny. 1987. The categorical abstract machine. *Science of computer programming* 8, 2 (1987), 173–202.

[13] P-L Curien. 1986. Categorical combinators. *Information and Control* 69, 1-3 (1986), 188–254.

[14] Conal Elliott. 2017. Compiling to categories. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 27.

[15] Andrzej Filinski. 2001. Normalization by evaluation for the computational lambda-calculus. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 151–165.

[16] Yves Lafont. 1988. The linear abstract machine. *Theoretical computer science* 59, 1-2 (1988), 157–180.

[17] Sam Lindley. 2005. Normalisation by evaluation in the compilation of typed functional programming languages. (2005).

[18] Saunders MacLane and Ieke Moerdijk. 1992. Sheaves in geometry and logic: a first introduction to topos theory. (1992).

[19] Conor McBride. 2018. Everybody's got to be somewhere. *Electronic Proceedings in Theoretical Computer Science* 275 (2018), 53–69.

[20] John C Mitchell and Eugenio Moggi. 1991. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic* 51, 1-2 (1991), 99–124.

[21] Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

[22] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything old is new again: quoted domain-specific languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial*

Evaluation and Program Manipulation. ACM, 25–36.

[23] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer.

[24] Russell O'Connor. 2017. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. ACM, 107–120.

[25] John C Reynolds. 1998. Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation* 11, 4 (1998), 363–397.

[26] Anne Sjerp Troelstra and Helmut Schwichtenberg. 2000. *Basic proof theory*. Number 43. Cambridge University Press.

[27] Nachiappan Valliappan, Solène Mirliaz, Elisabet Lobo Vesga, and Alejandro Russo. 2018. Towards Adding Variety to Simplicity. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 414–431.

A APPENDIX

A.1 Agda Implementation

The complete Agda implementation of the normalization procedure and mechanization of the proofs can be found at the URL https://github.com/nachivpn/expelim

A.2 Implementation of distributivity in BCC

```
Distr : BCC (a ∗ (b + c)) ((a ∗ b) + (a ∗ c))
Distr = apply • (pair
  (match
    (curry (inl • pair exr exl))
    (curry (inr • pair exr exl)) • exr)
  exl)
```