Thesis for The Degree of Licentiate of Philosophy

Be My Guest: Normalizing and Compiling Programs using a Host Language

Nachiappan Valliappan

CHALMERS

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2020

Be My Guest: Normalizing and Compiling Programs using a Host Language

Nachiappan Valliappan

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

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2020.

Abstract

In programming language research, *normalization* is a process of fundamental importance to the theory of computing and reasoning about programs. In practice, on the other hand, *compilation* is a process that transforms programs in a language to machine code, and thus makes the programming language a usable one. In this thesis, we investigate means of normalizing and compiling programs in a language using another language as the *host*. Leveraging a host to work with programs in a *guest* language enables reuse of features of the host that would otherwise be strenuous to develop.

The specific tools of interest are *Normalization by Evaluation* (NbE) and *Embedded Domain-Specific Languages* (eDSLs), both of which rely on a host language for their purposes. These tools are applied to solve problems in three different domains: Chapter 1 uses NbE to show that exponentials (or closures) can be eliminated from a categorical combinatory calculus; Chapter 2 uses NbE to propose a new proof technique based on normalization for showing noninterference; Chapter 3 uses eDSL techniques to enable the programming of resource-constrained IoT devices from Haskell.

Keywords: programming languages, normalization by evaluation, embedded domain-specific languages

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 beautiful path of theoretical pursuit? Or must I solve today's problems and push the boundaries of exciting 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 would like to thank him for his advice and guidance.

I would like to thank my friends and colleagues for helping me survive this cold, dark and strange place. I do not give you enough credit for pulling me away from my desk often enough. A list of names is in order here, but I will reserve this for a time that I write something worthy of your mention.

I should also acknowledge my family for their support in the years that lead to the beginning of my PhD. If it were not for them, I would not be in a position of luxury where the only things on my mind are the ones I choose to spend my time on.

The programming languages community is a wonderful lot of passionate people who care about tomorrow's researchers. I'm glad to be a part of it.

Contents

Introduction

A programming language is a formal language that consists of a term *syntax*, that specifies well-formed programs, and a *semantics*, that specifies how programs can be computed. The syntax of the language dictates the structure of the programs and provides a programming interface for developing software. The semantics, on the other hand, specifies the behavior of programs, and is used both as a formal tool to reason about programs and to guide the implementation of the language.

This thesis investigates the formalization and implementation of the syntax and semantics of a *guest* (or *object*) language by piggybacking on the syntax and semantics of a *host* language. Treating a language of interest as a guest with a familiar host, allows us to reuse features of the host language to work with the guest. The narrative of this thesis is bundled together by the tools it uses: *Normalization by Evaluation* (NbE) and *Embedded Domain-Specific Languages* (eDSLs). Both these tools use a host language to go about their business, and are closely related in the way they operate. These tools are used here to solve problems in three guest languages from entirely different domains.

Be my Guest

Before we get to the nuts and bolts of this thesis, let us take a brief introductory tour to understand the said tools and their importance in the development of programming languages. My main agenda here, in this section, is to

promote these tools as useful additions to the working computer scientist's toolkit for the purposes of normalization and compilation of programming languages. Now, if I may, allow me to preach a little.

Normalization by Evaluation In programming languages and mathematics, normalization is the process of transforming a complex expression to a simpler one by reducing it. Traditional normalization algorithms reduce a given expression by performing a series of reduction steps. An expression is rewritten several times before it can no longer be reduced, yielding a normal form of the expression. Normalization by Evaluation (NbE) [9, 26], on the other hand, bypasses rewriting entirely and instead normalizes an expression by evaluating it in a host language. NbE algorithms use a specialized interpreter to evaluate an expression in the host, and then convert the resulting value back to an expression in normal form by *reifying* them. *What is this sorcery?!* you may rightly wonder. Let me give you an example.

Let us suppose that we would like to normalize simple arithmetic expressions containing the addition of natural numbers. Using a constant symbol *Zero* and a successor function *Succ*, we may encode the natural numbers 0 as *Zero*, 1 as *Succ* 0, 2 as *Succ* (*Succ* 0), and so on. 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 involved any rewriting.

NbE achieves normalization in two steps: 1) *evaluating* the expressions in a host language, and 2) *reifying* the resulting values back to expressions. Let us implement NbE for our example using 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 -> Int
eval Zero     = 0
eval (Succ x) = eval x + 1
eval (x + y)  = eval x + eval y
```

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

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

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

```
norm :: Expr -> Expr
norm e = reify (eval e)
```

I insist that the reader convince themselves that an invocation of `norm` on the expression *Succ Zero* + *Succ* (*Succ Zero*) does indeed return its normal form *Succ* (*Succ* (*Succ* 0)). `norm` uses the ability of Haskell to evaluate the addition of integers to normalize the addition of natural numbers. This function can be modified easily to other arithmetic operators, and, with some care, even to support variables 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, 16, 21, 26], decide equality in algebraic structures [4], typecheck dependently-typed

programming languages [3, 22], and to prove completeness [5, 15] and coherence [10] theorems. NbE algorithms have been observed to yield much faster normalization than their rewriting counterparts [8, 25], and there is also evidence that indicates that it can be used to speed up compilation in optimizing compilers [25].

In this thesis, I use NbE as a theoretic tool to prove behavioral properties about two different programming calculi via normalization—a strategy which I shall call *proof by normalization*. NbE is not a requirement for this strategy, but rather a means. The main reason to use NbE is its practical convenience: when done in a host language, such as Agda, that functions both as a programming language and a proof assistant, it can be used to both reduce guest expressions and reason about its normal forms. NbE does not define normal forms with respect to a reduction relation, which also allows for a more flexible application-specific interpretation of what it means to be "normal". I bypass entirely the need to formulate a reduction relation or prove typical normalization results such as termination, confluence and strong normalization.

Embedded Domain-Specific Languages An 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 an example with arithmetic expressions again. The following library functions in Haskell constitute an eDSL to write simple arithmetic expressions.

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

```
(*) :: Expr -> Expr -> Expr
```

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

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

Using this function, we may simply 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 though the eDSL does not implement these features natively, we are able to use them to write expressions. In this fashion, eDSLs make it easy to derive additional functionality by leveraging the host language.

EDSLs, specifically in Haskell, have found a wide range of applications: hardware description [11], digital signal-processing [6], runtime verification [18, 29], parallel and distributed programming [13, 20], GPU programming [14]—and the list goes on. In this thesis, I use the eDSL technique as a means of providing a richer programming interface to a highly restrictive programming language for programming IoT applications.

Fistful of Nails: Problem Domains

This section gives an overview of the problems addressed in this thesis, and discusses the application of the said tools to these problems. The goal here is to discuss the interest in domains of these problems and provide sufficient background for the upcoming chapters (see Figure 1). Being entirely different domains, the following subsections may be read in any order.

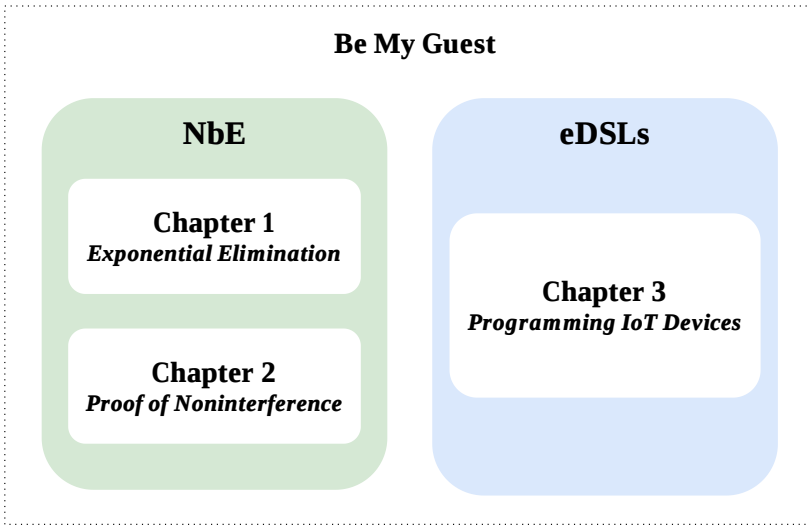


Figure 1: Outline of this thesis

Exponential Elimination for 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 β reduction 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) [17], 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 [1]. 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 [23].

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 [19] and in the development of a language for executing smart contracts on the blockchain [28].

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 [35]. In [19], 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.

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

Chapter 1 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 higher-order functions in functional calculus (known as *defunctionalization*) is not news [27], 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.

Proving Noninterference.

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 2 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 [31] in Haskell, which is a simply-typed lambda calculus extended with IFC primitives.

Programming IoT devices.

The Challenge. The Internet of Things (IoT) conceives a future where "things", physical objects, are interconnected over a network. The realisation of this future relies on the use of embedded electronic devices that facilitate communication over the network. Safe and effective programming of embedded electronic devices, however, is well known to be a difficult problem since these devices do not enjoy the same computational resources (such as power and memory) as a traditional computer.

Consider the Nordic Semiconductor *nrf52840DK*, which is a low power micro-controller, that we use in this thesis to build a sample application. It has a 64Mhz processor, 256KB RAM, and 1MB flash memory. Compare this with the specification of the MacBook Pro that I am writing this thesis on: 2.3 GHz Dual-Core processor, 8GM RAM, and 256 GB flash memory. Even the L3 *cache* of this computer is 4MB—which is *sixteen times* the amount of flash memory on the micro-controller! The advantage however, is that the micro-controller runs on a coin-sized battery that could keep it alive for days, while my MacBookPro cannot get through a day of work without being plugged.

So how do software developers cope with such low availability of resources? They use a low-level language such as C to fine tune their applications. The problem, however, is writing programs in C makes it hard to identify bugs and security vulnerabilities in the application. Even a seemingly innocent mistake might lead to severe vulnerabilities which could be exploited to perform malicious activity. For example, the *buffer overflow* problem is a classic exploit that illustrates the issue with manual memory management—a feature that is characteristic of C.

High-level Languages to the Rescue? On the other end of the spectrum of languages, we have high-level languages such as Java, Python, and Haskell. High-level languages free the programmer from a number of low-level operations that could cause security problems. For example, the Haskell runtime system uses a garbage collector to automatically manage memory, which means the programmer need not worry about buffer overflow attacks in most scenarios. Moreover, high-level languages also make it possible to implement a number of solutions to fine-grained analysis and control of the flow of sensitive information in a program [24, 30, 31, 32]. The price, however, is that these features require a powerful runtime system, which de-

mands much more computational resources than low-level languages. This means that, in most cases, high-level languages are not even an option for programming resource constrained devices.

Synchronous Programming, Embedded. Synchronous programming is a programming paradigm that is used to program *reactive* systems on embedded devices. Synchronous programming languages are well suited for reactive IoT applications since they provide a highly restrictive programming interface that is optimized for execution with a fixed amount of resources. In this thesis, we investigate ways to enrich the programming interface of a synchronous programming language, without demanding additional resources for execution.

Chapter 3 discusses the implementation of a synchronous language called Lustre [12] as an eDSL in Haskell, using novel embedding techniques that reuse Haskell features such as functions and pattern matching. Embedding Lustre in Haskell enables us to write programs that are more concise and less error-prone, while keeping the minimal nature of Lustre intact. The embedding also enables us to enforce security policies using IFC primitives that are reminiscent of the Haskell library LIO [32]. The eDSL is proposed as a solution to programming reactive IoT applications using Haskell.

Collection of Old Stories

Each chapter in this thesis has been developed in collaboration with a few others, and the contents have been published separately at different venues. Here is a quick rundown:

- Chapter 1 is based on [36], which was developed in collaboration with Alejandro Russo and appeared at the *21st Symposium on Principles and Practice of Declarative Programming (PPDP '19)*.
- Chapter 2 is based on [33], which was developed in collaboration with Carlos Tomé Cortiñas and appeared at the *14th Workshop on Programming Languages and Analysis for Security (PLAS '19)*.
- Chapter 3 is based on [34], which was developed in collaboration with Robert Krook, Alejandro Russo and Koen Claessen, and appeared at the *13th Haskell Symposium (Haskell '20)*.

This thesis 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.

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] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, 1987.

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

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

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

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

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

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

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

[20] 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.

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

[22] 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.

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

[24] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *Proc. of the IEEE Workshop on Computer Security Foundations (CSFW '06)*. IEEE Computer Society, 2006.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] A. Russo. Functional pearl: Two can keep a secret, if one of them uses Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015. ACM, 2015.

[31] 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.

[32] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*, 2011.

[33] C. Tomé Cortiñas and N. Valliappan. Simple Noninterference by Normalization. In *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, pages 61–72, 2019.

[34] N. Valliappan, R. Krook, A. Russo, and K. Claessen. Towards Secure IoT Programming in Haskell. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, pages 136–150, 2020.

[35] 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.

[36] N. Valliappan and A. Russo. Exponential Elimination for Bicartesian Closed Categorical Combinators. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*, pages 1–13, 2019.

1

Exponential Elimination for Bicartesian Closed Categorical Combinators

Nachiappan Valliappan, Alejandro Russo

PPDP '19, October 7–9, 2019, Porto, Portugal

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 chapter, 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, 17]. 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 [25] 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 [28], 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 [23]—also known as *defunctionalization* [26].

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* [21], that normal forms of terms are consistent with the conversion rules by showing that they are interconvertible. (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, 24]. 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: **1** (for unit), **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

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

[1] https://github.com/nachivpn/expelim

```
data _≈_ : BCC a b → BCC a b → Set where
  -- categorical rules
  idr         : f • id ≈ f
  idl         : id • f ≈ f
  assoc       : f • (g • h) ≈ f • (g • h)
  -- elimination rules
  exl-pair    : (exl • pair f g) ≈ f
  exr-pair    : (exr • pair f g) ≈ g
  match-inl   : (match f g • inl) ≈ f
  match-inr   : (match f g • inr) ≈ g
  apply-curry : apply • (curry f ⊗ id) ≈ f
  -- uniqueness rules
  uniq-init   : init ≈ f
  uniq-unit   : unit ≈ f
  uniq-pair   : exl • h ≈ f → exr • h ≈ g → pair f g ≈ h
  uniq-curry  : apply • h ⊗ id ≈ f → curry f ≈ h
  uniq-match  : h • inl ≈ f → h • inr ≈ g → match f g ≈ h
  -- equivalence and congruence rules
  refl        : f ≈ f
  sym         : f ≈ g → g ≈ f
  trans       : f ≈ g → g ≈ h → f ≈ h
  congl       : x ≈ y → f • x ≈ f • y
  congr       : x ≈ y → x • f ≈ y • f
```

Fig. 2. Conversion rules for BCC

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* : 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 uniq-curry refl.

```
eta⇒ : 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

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 $⟦\ a\ ⟧ \twoheadrightarrow ⟦\ b\ ⟧$, 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 : BCC a b) → t ≈ q (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*, 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.

```
drop iden : Sel ((a + b) ∗ c) (a + b)
keep (drop iden) : Sel (a ∗ b ∗ c) (a ∗ c)
drop (keep (drop iden)) : Sel (a ∗ b ∗ c ∗ d) (a ∗ c)
```

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.

[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.

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 : iden ∘ s ≡ s
sel-idr : s ∘ iden ≡ s
sel-assoc : (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

```
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

simple as exl or it could be:

```
apply • (pair (curry unit • exl) exr) : BCC (𝟏 ∗ 𝟏) 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.

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 : a ◁ b → a ◁ (b ⊗ c)
  subr : 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 [27]. 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.

[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 ($*$)

Theorem 5.1. All normal forms, as defined by the data type Nf, satisfy the weak subformula property.

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.

$$\text{case?} : \text{Ne}\ a\ (b + c) \to \text{Nf}\ (a * b)\ d \to \text{Nf}\ (a * c)\ d \to \text{Ne}\ a\ d$$

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?

$$\text{case?} : \text{Nf}\ a\ (b + c) \to \text{Nf}\ (a * b)\ d \to \text{Nf}\ (a * c)\ d \to \text{Nf}\ a\ d$$

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

```
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

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.

```
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 → Ne a b → 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.

$$
\begin{aligned}
⟦\ \mathbf{1}\ ⟧\mathsf{n} &= ⊤ \\
⟦\ \mathbf{0}\ ⟧\mathsf{n} &= ⊥ \\
⟦\ \mathsf{base}\ ⟧\mathsf{n} &= ?? \\
⟦\ t_1 * t_2\ ⟧\mathsf{n} &= ⟦\ t_1\ ⟧\mathsf{n} \times ⟦\ t_2\ ⟧\mathsf{n} \\
⟦\ t_1 + t_2\ ⟧\mathsf{n} &= ⟦\ t_1\ ⟧\mathsf{n} \uplus ⟦\ t_2\ ⟧\mathsf{n} \\
⟦\ t_1 \Rightarrow t_2\ ⟧\mathsf{n} &= ⟦\ t_1\ ⟧\mathsf{n} \to ⟦\ t_2\ ⟧\mathsf{n}
\end{aligned}
$$

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 × 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 → B .In i) to (A .In j → 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 [19]—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 [18].

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 ↠ Tree' P
join : Tree' (Tree' P) ↠ Tree' P
map : (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:

$$
\begin{aligned}
&\mathsf{R} : \{b : \mathsf{Ty}\} \to \mathsf{BCC}\ a\ b \to \mathsf{Sem}\ a\ [\![\ b\ ]\!] \to \mathsf{Set} \\
&\mathsf{R}\ \{\mathbf{1}\} \qquad t\ v = \top \\
&\mathsf{R}\ \{\mathsf{base}\} \qquad t\ v = t \approx \mathsf{q}\ v \\
&\mathsf{R}\ \{b_1 * b_2\} \quad t\ v = \mathsf{R}\ (\mathsf{exl} \bullet t)\ (\mathsf{proj}_1\ v) \times \mathsf{R}\ (\mathsf{exr} \bullet t)\ (\mathsf{proj}_2\ v) \\
&\mathsf{R}\ \{b_1 \Rightarrow b_2\}\ t\ v = (s : \mathsf{Sel}\ c\ \_) \\
&\quad \to \mathsf{R}\ t'\ x \to \mathsf{R}\ (\mathsf{apply} \bullet \mathsf{pair}\ (\mathsf{liftBCC}\ s\ t)\ t')\ (v\ s\ x) \\
&\mathsf{R}\ \{\mathbf{0}\} \qquad t\ v = \mathsf{R}_0\ t\ v \\
&\mathsf{R}\ \{b + c\} \qquad t\ v = \mathsf{R}_+\ t\ v
\end{aligned}
$$

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 : (Rl : 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 Sem a **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 inl • 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' → R t' v → R t v
```

Proof. By induction on the return type of t and t'. The proof is fairly straight-forward equational reasoning using the conversion rules (≈). The empty and sum types can be handled by induction on the tree. □

Lemma 7.2 (Lifting preserves relations)**.** If a term t : BCC a b is related to a value v : Sem a ⟦ b ⟧, then lifting the term is related to lifting the value, for any applicable selection s.

liftPresR : 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 • t' is related to the evaluation of t with the input v, for all terms t. That is, the fundamental theorem holds if 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 : 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 : (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 : BCC a b can be converted to an equivalent term f' : 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 : Nf a b resulting from the application norm f such that $f \approx$ q n. Since a and b are first-order types, we also have a DBC term qD n : DBC a b, which does not use exponentials by construction. Additionally, since the function qD is a restriction map of the function q, qD n must be equivalent to q n, and hence to f. This can be shown by proving that the embedding of the DBC term qD n into BCC is convertible to q n, and hence to f. Thus we have an equivalent DBC term f' = qD n. □

8 Simplicity, an application

Simplicity is a typed combinator language for programming smart contracts in blockchain applications [25]. 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.

```
size 1         = 0
size (t₁ * t₂) = size t₂ `+ size t₂
size (t₁ + t₂) = 1 `+ max (size t₁) (size t₂)
```

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. [28] 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:

$$\mathsf{size}\ (t_1 \Rightarrow t_2) = \mathsf{size}\ t_2\ \hat{}\ \mathsf{size}\ t_1$$

Valliappan et al. [28] 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 [28]:

```
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 [18]. 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 [20]. 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 ($\lambda_{\Rightarrow \mathbf{1} * +}$). Balat et al. [7] solve the *extensional* normalization problem using NbE for the STLC including the empty type ($\lambda_{\Rightarrow \mathbf{1} * + \mathbf{0}}$). Abel and Sattler [2] provide an account of NbE for $\lambda_{\Rightarrow \mathbf{1} * + \mathbf{0}}$ 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* [22], 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 ≈. 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 ≈, 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.

In Retrospect

The distributivity requirement in DBC is perhaps the most important takeaway from this work: it says that the target abstract machine must at least support distributivity if we do not want to implement closures.

A drawback of this work, however, is that the proof of correctness for the normalization procedure takes the full liberty of using all the equations in the calculus. This makes it hard to see exactly what equations are *required* to eliminate exponentials or how the size of the normal forms can be tamed. For example, are the uniqueness rules really required to eliminate exponentials? Answering this requires a more careful analysis of the NbE model and potentially further refinement.

Perhaps it would have been better to use a directed rewrite relation $\_ \longrightarrow^* \_$ in place of the symmetric conversion relation $\_ \approx \_$. This would have made it easier to see exactly how BCC terms are reduced in a more conventional sense. However, designing a satisfactory rewrite relation for categorical combinators (even in the absence of sums) has proven to be a difficult task [13, 16] with no definitive result to date—to the best of our knowledge.

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] Thérese Hardin. 1989. Confluence results for the pure strong categorical logic ccl. λ-calculi as subsystems of ccl. *Theoretical computer science* 65, 3 (1989), 291–342.

[17] Yves Lafont. 1988. The linear abstract machine. *Theoretical computer science* 59, 1-2 (1988), 157–180.

[18] Sam Lindley. 2005. Normalisation by evaluation in the compilation of typed functional programming languages. (2005).

[19] Saunders MacLane and Ieke Moerdijk. 1992. Sheaves in geometry and logic: a first introduction to topos theory. (1992).

[20] Conor McBride. 2018. Everybody's got to be somewhere. *Electronic Proceedings in Theoretical Computer Science* 275 (2018), 53–69.

[21] 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.

[22] Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

[23] 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.

[24] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer.

[25] 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.

[26] John C Reynolds. 1998. Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation* 11, 4 (1998), 363–397.

[27] Anne Sjerp Troelstra and Helmut Schwichtenberg. 2000. *Basic proof theory*. Number 43. Cambridge University Press.

[28] 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)
```

2

Simple Noninterference by Normalization

Carlos Tomé Cortiñas, Nachiappan Valliappan

PLAS'19, November 15, 2019, London, United Kingdom

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 chapter, 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* [3, 7, 27]. In this chapter, 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 chapter 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 et al. 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 $\mathsf{Bool}_{\mathrm{L}} \times \mathsf{Bool}_{\mathrm{H}} \to \mathsf{Bool}_{\mathrm{L}} \times \mathsf{Bool}_{\mathrm{H}}$. 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 chapter 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 et al. [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`:

data S $(\ell :: \mathit{Lattice})$ a

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

$$
\begin{aligned}
&\mathit{return} :: a \;\to\; S\ \ell\ a \\
&(\gg\!=) \;\; :: S\ \ell\ a \;\to\; (a \;\to\; S\ \ell\ b) \;\to\; S\ \ell\ b \\
&\mathit{up} \quad\;\; :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \;\Rightarrow\; S\ \ell_{\mathrm{L}}\ a \;\to\; S\ \ell_{\mathrm{H}}\ a
\end{aligned}
$$

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 (see [11]) {L, H} where the only disallowed flow is from secret (H) to public (L), denoted $\mathrm{H} \not\sqsubseteq \mathrm{L}$. The following program written using the `seclib` API is well-typed and—intuitively—secure:

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

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 chapter, 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 $\ell_{\mathrm{H}} \not\sqsubseteq \ell_{\mathrm{L}}$. Figure 1 defines the syntax of terms, types and contexts of λ_{sec}.

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

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 ($\gg=$), 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.

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

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

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

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]

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

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 [2, 15], 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.

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.

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

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

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

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

$$\gamma\text{-S}\ \frac{\begin{array}{c}\Gamma \vdash t_1 : \mathsf{S}\ \ell\ \tau_1 \\ \Gamma, x : \tau_1 \vdash t_2 : \mathsf{S}\ \ell\ \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash t_3 : \mathsf{S}\ \ell\ \tau_3\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{let}\ x = (\mathsf{let}\ y = t_1\ \mathsf{in}\ t_2)\ \mathsf{in}\ t_3 \approx \\ \mathsf{let}\ y = t_1\ \mathsf{in}\ (\mathsf{let}\ x = t_2\ \mathsf{in}\ t_3) : \mathsf{S}\ \ell\ \tau_3\end{array}}$$

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

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

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

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

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

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

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 $\Gamma \vdash_{\text{ne}} t : \tau$ (Figure 5) and normal forms as $\Gamma \vdash_{\text{nf}} t : \tau$ (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.

$\boxed{\Gamma \vdash_{\text{ne}} t : \tau}$

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash_{\text{ne}} x : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash_{\text{ne}} t : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash_{\text{nf}} s : \tau_1}{\Gamma \vdash_{\text{ne}} t\ s : \tau_2} \qquad \textsc{Fst}\ \frac{\Gamma \vdash_{\text{ne}} t : \tau_1 \times \tau_2}{\Gamma \vdash_{\text{ne}} \mathsf{fst}\ t : \tau_1}$$

$$\textsc{Snd}\ \frac{\Gamma \vdash_{\text{ne}} t : \tau_1 \times \tau_2}{\Gamma \vdash_{\text{ne}} \mathsf{snd}\ t : \tau_2}$$

Fig. 5. Neutral 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 $\Gamma \vdash_{\text{ne}} \tau$, neutrality states that the type τ must occur as a *subformula* of a type in the context Γ.

For instance, given a neutral form $\Gamma \vdash_{\text{ne}} n : \mathsf{Bool}$, 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 $\Gamma = [x : () \Rightarrow \mathsf{Bool}\ ,\ y : \mathsf{S}\ \ell_{\text{H}}\ \iota]$. 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

$\boxed{\Gamma \vdash_{\mathrm{nf}} t : \tau}$

$$\textsc{Unit}\ \frac{}{\Gamma \vdash_{\mathrm{nf}} () : ()} \qquad \textsc{Lam}\ \frac{\Gamma, x : \tau_1 \vdash_{\mathrm{nf}} t : \tau_2}{\Gamma \vdash_{\mathrm{nf}} \lambda\, x.t : \tau_1 \Rightarrow \tau_2} \qquad \textsc{Base}\ \frac{\Gamma \vdash_{\mathrm{ne}} t : \iota}{\Gamma \vdash_{\mathrm{nf}} t : \iota}$$

$$\textsc{Ret}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau}{\Gamma \vdash_{\mathrm{nf}} \mathsf{return}\ t : \mathsf{S}\ \ell\ \tau}$$

$$\textsc{LetUp}\ \frac{\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \qquad \Gamma \vdash_{\mathrm{ne}} t : \mathsf{S}\ \ell_{\mathrm{L}}\ \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\mathrm{nf}} s : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau_2}{\Gamma \vdash_{\mathrm{nf}} \mathsf{let}{\uparrow}\ x = t\ \mathsf{in}\ s : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau_2}$$

$$\textsc{Left}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau_1}{\Gamma \vdash_{\mathrm{nf}} \mathsf{left}\ t : \tau_1 + \tau_2} \qquad \textsc{Right}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau_2}{\Gamma \vdash_{\mathrm{nf}} \mathsf{right}\ t : \tau_1 + \tau_2}$$

$$\textsc{Case}\ \frac{\Gamma \vdash_{\mathrm{ne}} t : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash_{\mathrm{nf}} t_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash_{\mathrm{nf}} t_2 : \tau}{\Gamma \vdash_{\mathrm{nf}} \mathsf{case}\ t\ (\mathsf{left}\ x_1 \to t_1)\ (\mathsf{right}\ x_2 \to t_2) : \tau}$$

Fig. 6. Normal forms.

- 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 [2, 5] 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 et al. [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

[4] λ_{sec} does not have polymorphic types, in this case τ represents an arbitrary but concrete type, for instance unit ().

words, it states that varying the secret input must *not interfere* with the public output.

As explained before, for $\emptyset \vdash f : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau \Rightarrow \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool}$ 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 $\emptyset \vdash_{\mathrm{nf}} f : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau \Rightarrow \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool}$, either $f \equiv \lambda\, x.(\mathsf{return}\ \mathit{true})$ or $f \equiv \lambda\, x.(\mathsf{return}\ \mathit{false})$

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 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 $\lambda_{\mathbf{sec}}$ 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:

$$\begin{aligned}&\mathsf{eval} : (\Gamma \vdash \tau) \to (⟦\ \Gamma\ ⟧ \to ⟦\ \tau\ ⟧)\\ &\mathsf{reify} : (⟦\ \Gamma\ ⟧ \to ⟦\ \tau\ ⟧) \to (\Gamma \vdash_{\mathrm{nf}} \tau)\end{aligned}$$

In these types, the function $⟦\ \_\ ⟧$ interprets types and contexts in λ_{sec} as types in the host language. That is, the type $⟦\ \tau\ ⟧$ denotes the interpretation of the (λ_{sec}) type τ in the host language, and similarly for $⟦\ \Gamma\ ⟧$. On the other hand, the function $⟦\ \Gamma\ ⟧ \to ⟦\ \tau\ ⟧$—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.1. 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:

```
eval x          γ = lookup x γ
eval ()         γ = tt
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 $\mathsf{S}\ \ell\ \tau$, 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:

$$
\begin{aligned}
&\mathsf{return} : a \to T\ \ell\ a \\
&\mathsf{bind}\ \ : T\ \ell\ a \to (a \to T\ \ell\ b) \to T\ \ell\ b \\
&\mathsf{up}\ \ \ : (\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}) \to T\ \ell_{\mathrm{L}}\ a \to T\ \ell_{\mathrm{H}}\ a
\end{aligned}
$$

To satisfy this specification, we define the data type T in Agda with the following constructors:

$$
\textsc{Return}\ \frac{x : a}{\mathsf{return}\ x : T\ \ell\ a} \qquad \textsc{BindN}\ \frac{p : \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \qquad n : \mathsf{Ne}\ \mathsf{S}\ \ell_{\mathrm{L}}\ \tau \qquad f : \mathsf{Var}\ \tau \to T\ \ell_{\mathrm{H}}\ a}{\mathsf{bindNe}\ p\ n\ f : T\ \ell_{\mathrm{H}}\ a}
$$

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:

$$
\begin{aligned}
&\mathsf{up}\ p\ (\mathsf{return}\ v)\ = \\
&\quad \mathsf{return}\ v \\
&\mathsf{up}\ p\ (\mathsf{bindNe}\ q\ n\ f)\ = \\
&\quad \mathsf{bindNe}\ (\mathsf{trans}\ q\ p)\ n\ (\lambda\ x \to \mathsf{up}\ p\ (f\ x))
\end{aligned}
$$

To understand this implementation, suppose that $p : \ell_{\mathrm{M}} \sqsubseteq \ell_{\mathrm{H}}$ for some labels ℓ_{M} and ℓ_{H}. A monadic value of type $T\ \ell_{\mathrm{M}}\ a$ which is constructed by a return can be simply re-labeled to $T\ \ell_{\mathrm{H}}\ a$ since return can be used to construct a monadic value on any label. For the case of $\mathsf{bindNe}\ q\ n\ f$, we have that $q : \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{M}}$ and $n : \mathsf{Ne}\ \mathsf{S}\ \ell_{\mathrm{L}}\ \tau_1$, hence $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ 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 $T\ \ell_{\mathrm{H}}\ a$.

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:

$$\begin{aligned}
&\mathsf{eval}\ (\mathsf{return}\ t)\ \gamma \;=\; \mathsf{return}\ (\mathsf{eval}\ t\ \gamma)\\
&\mathsf{eval}\ (\mathsf{up}\ p\ t)\ \ \gamma \;=\; \mathsf{up}\ p\ (\mathsf{eval}\ t\ \gamma)\\
&\mathsf{eval}\ (\mathsf{let}\ x = t\ \mathsf{in}\ s)\ \gamma \;=\\
&\quad \mathsf{bind}\ (\mathsf{eval}\ t\ \gamma)\ (\lambda\ v\ \to\ \mathsf{eval}\ s\ (\gamma\ [x \mapsto v]))
\end{aligned}$$

For implementing reflection, we can use bindNe to register a neutral binding and recursively reflect the given variable:

$$\begin{aligned}
&\mathsf{reflect}\ \{\mathsf{S}\ \ell\ \tau\}\ n \;=\\
&\quad \mathsf{bindNe}\ \mathsf{refl}\ n\ (\lambda\ x\ \to\ \mathsf{return}\ (\mathsf{reflect}\ \{\tau\}\ x))
\end{aligned}$$

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:

$$\begin{aligned}
&\mathsf{reifyVal}\ \{\mathsf{S}\ \ell\ \tau\}\ (\mathsf{return}\ v) \;=\\
&\quad \mathsf{return}\ (\mathsf{reifyVal}\ \{\tau\}\ v)\\
&\mathsf{reifyVal}\ \{\mathsf{S}\ \ell\ \tau\}\ (\mathsf{bindNe}\ \{p\}\ n\ f) \;=\\
&\quad \mathsf{let}{\uparrow}\ \{p\}\ x = n\ \mathsf{in}\ \mathsf{reifyVal}\ \{\tau\}\ (f\ x)
\end{aligned}$$

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 $\ell_{\mathrm{H}} \not\sqsubseteq \ell_{\mathrm{L}}$ and a function $\emptyset \vdash f : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau \Rightarrow \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool}$ then $\forall\ s_1\ s_2 : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau.$ $f\ s_1 \approx f\ s_2$

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 $\emptyset \vdash f : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau \Rightarrow \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool}$ 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 $\emptyset \vdash f : \mathsf{S}\ \ell_{\mathrm{H}}\ \tau \Rightarrow \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool}$, 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.

$$\textit{Substitution}\ \sigma ::= \sigma_\emptyset \mid \sigma\,[x \mapsto t]$$

$$\boxed{\Gamma \vdash_{\text{sub}} \sigma : \Delta}$$

$$\frac{\Gamma \vdash_{\text{sub}} \sigma : \Delta \qquad \Gamma \vdash t : \tau}{\Gamma \vdash_{\text{sub}} \sigma\,[\,x \mapsto t\,] : \Delta, x : \tau} \qquad \frac{}{\Gamma \vdash_{\text{sub}} \sigma_\emptyset : \emptyset}$$

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 $\ell_{\text{H}} \not\sqsubseteq \ell_{\text{L}}$, 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 $\sigma_2 \equiv \sigma_{\ell_L} ; \sigma^2_{\ell_H}$

[5]In Section 2 we purposely left capture-avoiding substitutions underspecified, we amend that here.

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:

$$\mathsf{ren} : \{\Gamma \leqslant \Delta\} \to (\Gamma \vdash_{\text{nf}} \tau) \to (\Delta \vdash_{\text{nf}} \tau)$$

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 $\Gamma \vdash_{\text{nf}} n : \tau$, 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** ($\Gamma \vdash_{\text{nf}} () : ()$). The normal form $()$ is constant.
- **Case 2** ($\Gamma \vdash_{\text{nf}} n : \iota$). 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 $\Gamma \vdash_{\text{ne}} \tau$, 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** ($\Gamma \vdash_{\text{nf}} \text{return } n : \text{S } \ell\ \tau$). 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 $\text{ren } (n') \equiv n$. By congruence of the relation $\equiv$, we get that $\text{return } (\text{ren } (n')) \equiv \text{return } n$. Note that the function ren is defined as $\text{ren } (\text{return } n') \equiv \text{return } (\text{ren } n')$, and hence by transitivity of $\equiv$, we have that $\text{ren } (\text{return } (n')) \equiv \text{return } n$. Thus, the normal form $\text{return } n$ is also constant.
- **Case 4** ($\Gamma \vdash_{\text{nf}} \text{let}\uparrow\ x = n \text{ in } m : \text{S } \ell_2\ \tau_2$). For this case, we have a neutral $\Gamma \vdash_{\text{ne}} n : \text{S } \ell_1\ \tau_1$ 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 $\text{S } \ell_2\ \tau$ 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** ($\Gamma \vdash_{\text{nf}} \text{left } n : \tau_1 + \tau_2$). Similar to return.
- **Case 6** ($\Gamma \vdash_{\text{nf}} \text{right } n : \tau_1 + \tau_2$). 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 $\ell_H \not\sqsubseteq \ell_L$, 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 $\ell_H \not\sqsubseteq \ell_L$; 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 Δ, $\Delta \vdash t\ [\ \sigma_{\ell_L}\ ] : \tau$. 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 [7, 23]. 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 $x : \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool} \times \mathsf{S}\ \ell_{\mathrm{H}}\ \mathsf{Bool} \vdash \mathsf{snd}\ t : \mathsf{S}\ \ell_{\mathrm{H}}\ \mathsf{Bool}$:

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 chapter 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 et al. [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* [20, 31], 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.

In Retrospect

The unsettling part of this work is the simplicity of the calculus used to illustrate this proof technique.

> *"My only qualm with the paper is that it presents itself as a simpler alternative to dynamic/denotational techniques, but at the same time only applies itself to a simpler context: a normalizing language."*
> —An anonymous reviewer on this work

It is not shown whether this technique is applicable in the presence of general recursion. But as mentioned earlier, it seems that full $\beta\eta$ normalization is not a requirement for this technique, and thus it may well be possible. Intuition makes a strong case, but much remains to be seen in this direction.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G Riecke. 1999. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 147–160.

[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] Maximilian Algehed and Jean-Philippe Bernardy. 2019. Simple Noninterference from Parametricity. (2019).

[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] 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.

[6] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalization by evaluation. In *Prospects for Hardware Foundations*. Springer, 117–137.

[7] William J Bowman and Amal Ahmed. 2015. Noninterference for free. *ACM SIGPLAN Notices* 50, 9 (2015), 101–113.

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

[9] Olivier Danvy. 1998. Type-directed partial evaluation. In *DIKU International Summer School*. Springer, 367–411.

[10] Olivier Danvy, Morten Rhiger, and Kristoffer H Rose. 2001. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming* 11, 6 (2001), 673–680.

[11] Dorothy E Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (1976), 236–243.

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

[13] GA Kavvos. 2019. Modalities, cohesion, and information flow. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 20.

[14] Peng Li and Steve Zdancewic. 2010. Arrows for secure information flow. *Theoretical computer science* 411, 19 (2010), 1974–1994.

[15] Sam Lindley. 2005. Normalisation by evaluation in the compilation of typed functional programming languages. (2005).

[16] Conor McBride. 2018. Everybody's got to be somewhere. *Electronic Proceedings in Theoretical Computer Science* 275 (2018), 53–69.

[17] Kenji Miyamoto and Atsushi Igarashi. 2004. A modal foundation for secure information flow. In *Workshop on Foundations of Computer Security*. 187–203.

[18] Eugenio Moggi. 1989. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. IEEE, 14–23.

[19] Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

[20] Dominic A Orchard and Tomas Petricek. 2014. Embedding effect systems in Haskell. (2014).

[21] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

[22] Gordon Plotkin. 1973. *Lambda-definability and logical relations*. Edinburgh University.

[23] Alejandro Russo, Koen Claessen, and John Hughes. 2009. A library for light-weight information-flow security in Haskell. *ACM Sigplan Notices* 44, 2 (2009), 13–24.

[24] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *ACM Sigplan Notices*, Vol. 46. ACM, 95–106.

[25] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe haskell. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 137–148.

[26] Anne Sjerp Troelstra and Helmut Schwichtenberg. 2000. *Basic proof theory*. Number 43. Cambridge University Press.

[27] Stephen Tse and Steve Zdancewic. 2004. Translating dependency into parametricity. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 115–125.

[28] Nachiappan Valliappan and Alejandro Russo. 2019. Exponential Elimination for Bicartesian Closed Categorical Combinators. (2019).

[29] Marco Vassena and Alejandro Russo. 2016. On formalizing information-flow control libraries. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 15–28.

[30] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. 2018. Mac a verified static information-flow control library. *Journal of logical and algebraic methods in programming*

95 (2018), 148–180.

[31] Philip Wadler. 1998. The marriage of effects and monads. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 63–74.

A Appendix

A.1 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:

$$\textsc{Leaf}\ \frac{x : a}{\mathsf{leaf}\ x : D\ a}$$

$$\textsc{Branch}\ \frac{n : \mathsf{Ne}\ (\tau_1 + \tau_2) \qquad f : \mathsf{Var}\ \tau_1 \to D\ a \qquad g : \mathsf{Var}\ \tau_2 \to D\ a}{\mathsf{branch}\ n\ f\ g : D\ a}$$

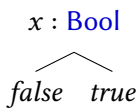
The leaf constructor constructs a leaf of the tree from a semantic value, while the branch constructor constructs a tree which represents a suspended decision over the value of a sum type. The branch constructor is the semantic equivalent of case in normal forms.

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 $D \llbracket \text{Bool} \rrbracket$ 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:

```
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 $D \llbracket \tau \rrbracket$, we need a function mkDec : $D \llbracket \tau \rrbracket \to \llbracket \tau \rrbracket$. 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 (Nf τ) $\to$ Nf τ. 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))
```

[6] We ignore the argument (as $\lambda \_$) here since it has the uninteresting type ()

```
  where
    match : (⟦ τ₁ ⟧ ⊎ ⟦ τ₂ ⟧) → ⟦ τ ⟧
    match (inj₁ v) = eval t₁ (γ [x₁ ↦ v])
    match (inj₂ v) = eval t₂ (γ [x₂ ↦ 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.

3

Towards Secure IoT Programming in Haskell

Nachiappan Valliappan, Robert Krook, Alejandro Russo, Koen Claessen

Haskell '20, August 27, 2020, Virtual Event, USA

Abstract. IoT applications are often developed in programming languages with low-level abstractions, where a seemingly innocent mistake might lead to severe security vulnerabilities. Current IoT development tools make it hard to identify these vulnerabilities as they do not provide end-to-end guarantees about how data flows *within and between* appliances. In this work we present Haski, an embedded domain specific language (eDSL) in Haskell for secure programming of IoT devices. Haski enables developers to write Haskell programs that generate C code without falling into many of C's pitfalls. Haski is designed after the synchronous programming language Lustre, and sports a backwards compatible information-flow control extension to restrict how sensitive data is propagated and modified within the application. We present a novel eDSL design which uses recursive monadic bindings and allows a natural use of functions and pattern matching to write embedded programs. To showcase Haski, we implement a simple smart house controller where communication is done via low-energy Bluetooth on the Zephyr IoT OS.

1 Introduction

The Internet of Things (IoT) conceives a future where "things" (embedded electronics) can be interconnected. While a compelling vision, recent events have demonstrated the *high vulnerability* of IoT (e.g., [5, 18, 34, 39]). Hence, it has become important to develop security solutions which address the concerns of unauthorized access to data and privacy loss.

We believe there are two major aspects which contribute to the current poor state-of-the-art in IoT security: *the chosen programming languages for development* and *the lack of end-to-end guarantees*. IoT development is often done in programming languages (like C) with low-level of abstractions, where a seemingly innocent mistake might lead to severe vulnerabilities like buffer overflows. Similarly, development tools present no end-to-end guarantees about how data flows *within and between* devices—thus making it hard to *confine* sensitive information.

Figure 1 shows the running example throughout this chapter: a simplified *smart house controller* called Halexa. Halexa consists of a micro-controller with Wifi access (required to fetch software updates) which is connected to three Bluetooth devices: a thermometer, a motion sensor, and a window. The micro-controller software is responsible for opening the window when it is

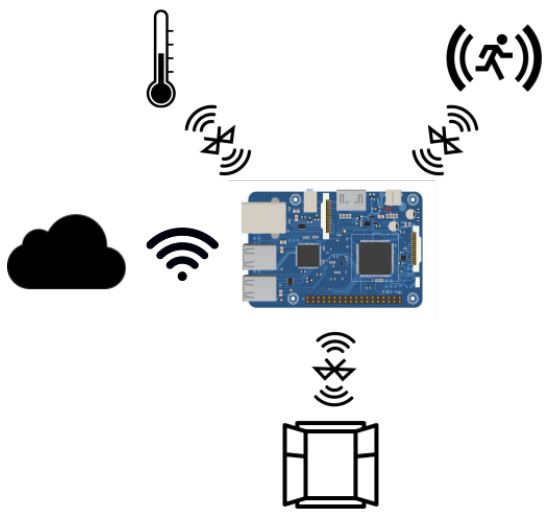


Fig. 1. A Smarthouse example

too hot inside the house. We assume that there is no Air Conditioning in the house—not an uncommon assumption in, for example, Nordic countries. While simple, this scenario presents interesting security and safety concerns: (i) to avoid robbery, windows must only be opened when there is someone at home, and (ii) the motion sensor data should be kept *confined* within the system and not leaked via Internet—leaking it can hint burglars about the vacancy of the house. Observe that the micro-controller needs to have access to the sensors' data in order to deliver its function. Can we use Haskell to program constrained devices and ensure the mentioned security requirements by construction?

In this chapter, we present Haski, an embedded domain specific language in Haskell for secure programming of IoT devices. Haski enables developers to write Haskell programs that generate C code without falling into many of C's pitfalls (e.g., those related to memory safety, undefined behavior, etc.). Haski follows the footsteps of the synchronous programming language Lustre [12, 20], which is an event-driven programming language with strong guarantees on resource usage—a must when programming low-power devices often found in IoT systems. Haski enhances Lustre with confidentiality and integrity security guarantees, as well as a means of communicating with streams generated by callback functions.

By adopting a synchronous programming model, Haski is able to provide resource bounds while removing memory-based security vulnerabilities by construction. Haski's design and implementation is unique compared with previous Haskell eDSLs for Lustre-like languages [7, 21]. Firstly, Haski presents a novel monadic design which allows programmers to leverage Haskell's monadic bindings (i.e., **do** and **mdo**) to specify streams as literate as possible. Secondly, Haski conceives a new DSL technique to compile Haskell functions on Haski-expressions into callable components of the target language. Finally, Haski provides user-defined enumeration types, where developers can simply use Haskell's **case** expression to inspect them, while raising a type-error in case of non-exhaustive patterns—thus making the code more robust. To address end-to-end guarantees, Haski incorporates information-flow control (IFC) techniques [33] to restrict how data propagates and gets modified—thus protecting the confidentiality and integrity of data. With IFC, developers can, for instance, incorporate third-party Haski code to analyze sensitive data like

that coming from the motion sensor while still preventing data leaks. To keep the types in eDSL simple, Haski enforces IFC at code-generation time by only tracking data propagation among end-points streams indicated by developers, e.g., the thermometer, motion sensor, window and Internet communication channel in Figure 1.

Contributions The main research contribution of this chapter is the design and implementation of Haski. We show how to design a synchronous language that is type-safe, protects confidentiality and integrity of data, handles I/O, and generates C code. Importantly, our design does not require any modifications to GHC or the use of compiler plug-ins. Instead, Haski uses embedding techniques by leveraging advanced type-level features of GHC such as GADTs [26], data kinds [40], existential types, and pattern synonyms [27]. Some of the techniques developed for Haski can be generalized and used for general DSL design in Haskell.

2 Haski by Example

Haski programs are written in Haskell using a special set of combinators. In this section, we illustrate various examples of Haski programs and showcase these combinators. For the upcoming examples, we use the data type *Action* to represent an action indicating whether our user Octavius has left (or entered) the house.

data *Action* = *Left* | *Entered*

The purpose of the *Action* data type (instead of, for example, *Bool*) is to illustrate the use of user-defined data types in Haski programs.

Recursive definitions A Haski program is a collection of stream definitions written in the *Haski* monad. A simple stream can be defined using the *letDef* combinator, which has the following type.

letDef :: *Stream a* → *Haski* (*Stream a*)

Using Haskell's **do** notation, we can use this combinator to bind streams to variables as follows.

```
left :: Haski (Stream Action)
left = do
  x ← letDef (val Left)
  return x
```

This program defines the constant stream that repeats the action *Left* as *Left*, *Left*, *Left*, … using the combinator $val :: HT\ a \Rightarrow a \rightarrow Stream\ a$. The type constraint *HT* ensures that a type is recognized by the Haski compiler and can be compiled by it. In this case, we may suppose that *Action* already satisfies this constraint, but we will later see how this is made possible.

Streams may also be defined recursively using the *fby* combinator (read *followed by*).

$$fby :: HT\ a \Rightarrow a \rightarrow Stream\ a \rightarrow Haski\ (Stream\ a)$$

The stream v ‘fby‘ s begins with the value v and is followed by the stream s. For example, we can define a stream of alternating actions such as *Left*, *Entered*, *Left*, *Entered*, … using the *fby* combinator as follows.

```
alt :: Haski (Stream Action)
alt = mdo
  x ← Left `fby` y
  y ← Entered `fby` x
  return x
```

The stream x here defines a stream that begins with *Left* and is followed by y. Similarly, y begins with *Entered* and is followed by x. We use the keyword **mdo**[1] instead of **do** for (mutually) recursive definitions.

Pattern matching definitions Streams can also be defined by pattern matching on values of other streams using the *match* combinator.

$$match :: (FinEnum\ a, Streams\ b) \Rightarrow Stream\ a \rightarrow (a \rightarrow b) \rightarrow Haski\ b$$

The combinator application $match\ e\ f$ defines the streams resulting from applying the observed value of e to f. The definition of f enables pattern matching on the value of e. The type constraint *FinEnum* subjects the type a to be *finitely enumerable*, and the constraint *Streams* overloads the type

[1] Enabled by the RecursiveDo extension

b to allow the function f to return multiple streams such as lists or tuples of streams. The constraint *FinEnum* ensures that *match* can only be used to pattern match on streams with finitely many values—a restriction which later enables code generation.

To illustrate the use of *match*, let us implement a simple cache mechanism that accepts requests to read and write actions, and responds with the last-written action, beginning with *Left*. Let us represent the request protocol using the data type *Req*.

```
data Req = Read | Write Action
```

Evidently, *Req* is finitely enumerable since it has only three possible values: *Read*, *Write Left*, and *Write Entered*. Hence we may use *match* on a stream *req* :: *Stream Req* as follows.

```
...
resp ← req `match` λcase
  Read → state
  Write x → val x
state ← Left `fby` resp
...
```

We shall use ellipses (…) in the code to hide the parts that are not relevant to the point being made. The response stream *resp* is defined by matching against the request stream *req*, where the second argument is a lambda-expression which pattern matches on its argument. We write **λcase** instead of $\lambda x \rightarrow$ **case** x **of**…[2].

The combinator *match* allows us to leverage the benefits of pattern matching in Haskell (such as variable binding, wild cards, guards, etc.) to generate code with simpler branching operators in the target language. For example, the definition of *resp* which pattern matches on *req* in the previous example, generates the following C code.

```
switch (req) {
  case READ      → resp = ...
  case WLEFT     → resp = ...
```

[2]Enabled by the LambdaCase extension

```
  case WENTERED → resp = ...
}
```

The cases are representative of the C values generated for the Haskell values of type *Req*.

A pattern match performed using *match* must handle all possible cases, and is enforced by the Haski compiler. If we leave out one of the cases in the above example, the Haski compiler throws an error such as the following—with line-numbers!

```
ghci> compile ...
*** Exception: Cache.hs:(20,18)-(21,22):
Non-exhaustive patterns in case
```

Nodes The stream *req* in the previous example has not been defined locally, and we wish for it to be a variable which can be substituted for by different contexts. *Nodes* allow us to define subprograms that abstract over stream expressions such as *req*, and thus enable an external caller to supply them. In Haski, nodes are written as Haskell functions, as shown below.

```
cache :: Stream Req → Haski (Stream Action)
cache = node "cache" $ λreq → mdo
  resp ← req `match` λcase
    Read → state
    Write x → val x
  state ← Left `fby` resp
  return resp
```

A node is created using the *node* combinator by providing a name string and a function as arguments.

```
node :: (Arg a, Box b) ⇒ String → (a → b) → (a → b)
```

The name string is used to identify a node uniquely during compilation, and the function defines the body of the node. The type constraints *Arg* and *Box* together ensure that the function $a \to b$ accepts streams as arguments and produces a stream result in the *Haski* monad, i.e., has a type of the shape $Stream\ a' \to Stream\ b' \to \ldots. \to Haski\ (Stream\ res)$.

Notice that the function which defines a node itself need not be inside the *Haski* monad as *Haski* (*Stream* $a' \to$ *Stream* $b' \to \ldots \to$ *Stream res*). This allows for a more natural type to be assigned to a node, and for them to be called and used as regular Haskell functions without any special combinators. For example, we may map over a list of streams as *mapM cache* (*requests* :: [*Stream Req*]) to generate a list of responses, each corresponding to a call of the node *cache*.

Compiling the node *cache* generates code which resembles the following in C.

```
typedef unsigned short Enum;
struct cache_mem {Enum action; };
Enum cache_step (struct cache_mem * self, Enum req) {
  ...
  return resp;
}
```

We shall return to the specifics later, but for now we simply observe that the node *cache* is compiled to a C function with an additional argument *self*. This argument maintains the internal state of the returned stream, which in this case is the last-written action. Also note that both the types (*Req* and *Action*) have been compiled to values of type *Enum*, which represents a small positive integer—a simplifying assumption made for all finitely enumerable types.

Primitive types and operators The Haski compiler supports standard primitive types of fixed size such as *Bool*, *Int*, etc.

```
instance HT Bool where ...
instance HT Int where ...
  -- similarly for other primitive types
```

The luxury of pattern-matching is limited to finitely enumerable types. Now suppose that we wish to adapt our cache example to a read and write integers instead of actions. Integers are not considered to be finitely enumerable for practical reasons, which means that we cannot use a Haskell data type with an integer in it for pattern matching. Instead, we must separate the request from the integer *payload* into two separate streams as follows.

data $Req_i = Read \mid Write$

$cache_i :: Stream\ Req_i \rightarrow Stream\ Int \rightarrow Haski\ (Stream\ Int)$
$cache_i = \ldots$

To program streams whose types are not finitely enumerable, we resort to the primitive operators supported by the compiler. Haski supports a fixed set of operators that are recognized by the target environment. These operators are overloaded when possible (e.g., +, ∗, etc.) and provided separately otherwise (e.g., *gtE*).

$(+) :: Stream\ Int \rightarrow Stream\ Int \rightarrow Stream\ Int$
$(*) :: Stream\ Int \rightarrow Stream\ Int \rightarrow Stream\ Int$
$gtE :: Stream\ Int \rightarrow Stream\ Int \rightarrow Stream\ Bool$
…

Sampling operators In addition to primitive operators, Haski also supports *sampling* operators called *when* and *merge* (from Lustre) for projecting and combining streams.

$when :: FinEnum\ b \Rightarrow Stream\ a \rightarrow (Stream\ b, b) \rightarrow Stream\ a$
$merge :: FinEnum\ a \Rightarrow Stream\ a \rightarrow (a \rightarrow Stream\ b) \rightarrow Stream\ b$

The operator *when* allows us to project streams to slower ones: the stream s_1`` `when` ``(s_2, x) produces the value of s_1 only when the value of s_2 is x. Operator *merge*, on the other hand, is a restrictive version of *match* that requires the streams returned by the function argument to be mutually complementary (i.e., at most one stream must produce a value at a time). As we will see in the next section, *merge* is in fact used to implement *match*.

Labeling primitives Streams which contain sensitive information can be labeled with a sensitivity level. Labeled streams are given the type *LStream a*, and may be understood as streams wrapped in a secure container whose access is controlled using specific primitives. A stream can be labeled and unlabeled using the primitives *label* and *unlabel* respectively, and the label of a stream can be queried using the *labelOf* primitive.

$label$ :: $Label \to Stream\ a \to Haski\ (LStream\ a)$
$unlabel$ :: $LStream\ a \to Haski\ (Stream\ a)$
$labelOf$ :: $LStream\ a \to Haski\ Label$

To understand the use of these primitives, let us implement a new version of the *cache* node where the request and response have been labeled. One reason to do this may be because we wish to keep the actions of a user of our system confidential. To implement the same behavior as before, we must now use the labeling primitives explicitly to label and unlabel the streams.

$secCache$ :: $LStream\ Req \to Haski\ (LStream\ Action)$
$secCache = node$ "secCache" \$ $\lambda req_l \to$ **do**
  $resp \leftarrow unlabel\ req_l \gg\!= cache$
  $\ell \leftarrow labelOf\ req_l$
  $resp_l \leftarrow label\ \ell\ resp$
  $return\ resp_l$

The code above unlabels the stream req_l as $unlabel\ req_l$. This raises the senstivity level of the program *secCache* to the label of req_l (also known as *tainting*), which forces all subsequently labeled streams (like $resp_l$) to be at least as sensitive as req_l. The sensitivity level of the program is then used by an administrator to enforce security policies on the program during compilation—as we shall see in Section 5.

3 Overview of Haski Compiler

Haski at its core is an embedding of Lustre in Haskell with support for IFC. This means that Haski enables the use of Haskell as a host language to write Lustre programs. A Lustre program, much like Haski, is a system of stream bindings accompanied by a collection of nodes invoked by them. Compiling a Haski program first builds a Lustre program, and then compiles it to C—thus generating low-level code as in the examples of the previous section.

The compilation function *compile*, which has the type $HT\ a \Rightarrow Haski\ (Stream\ a) \to IO\ ()$, compiles a Haski program and generates C code as a side-effect. Compilation builds a "main" node for the given program, which then acts as the point of invocation for the entire program. Note that the program is restricted to producing an output whose type satisfies the *HT*

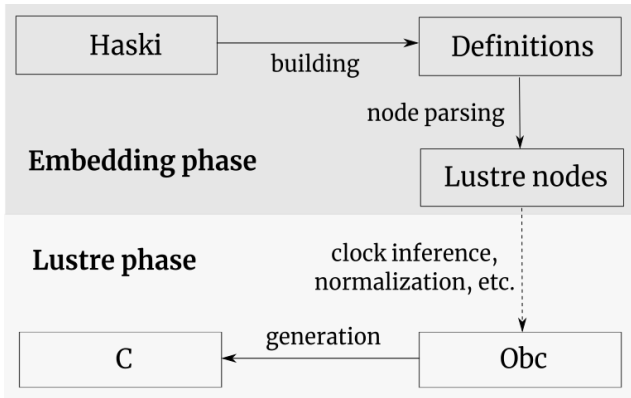


Fig. 2. Phases of eDSL compilation. The dashed arrow denotes a sequence of well-known compilation passes used to compile Lustre nodes [6].

constraint. This means that, although the program may use any Haskell types, its result must be of a type supported by the target language. This restriction, in combination with similar type constraints on the combinators, ensures that the use of Haskell's features that are not supported by the target environment (such as higher-order functions) are "evaluated away" during compilation time.

The compilation of a Haski program is achieved in two phases (see Figure 2): the Embedding phase constructs a list of Lustre nodes from a Haski program, and the Lustre phase then compiles the nodes to C functions. The first phase is implemented using a combination of deep and shallow embedding techniques, and consists of the compilation passes *building* and *node parsing*. The second phase, on the other hand, transforms Lustre nodes to C functions via an intermediate object-oriented language called Obc. This phase involves a sequence of compilation passes such as clock inference, normalization and scheduling, that are well-known in Lustre compilers [6].

The Lustre phase is implemented using a modular clock-directed compilation approach that is well-studied and has even been formally verified [1, 8]. We implement the passes in this phase by repeatedly traversing the abstract syntax tree of Lustre nodes and annotating it with the result of each phase (following Najd and Jones [25]). Our implementation of this phase is a straightforward adaptation of earlier work, and we do not discuss the details

```
data HaskiSt = HaskiSt { defs :: [Def], ... }
type Haski   = State HaskiSt

data Def where
  Let :: HT a ⇒ Var a → Stream a → Def
  Arg :: HT a ⇒ String → Var a → Stream a → Def
  Res :: HT a ⇒ String → Var a → Stream a → Def

data Stream a where
  Var    :: HT a ⇒ Var a → Stream a
  Val    :: HT a ⇒ a → Stream a
  Fby    :: HT a ⇒ a → Stream a → Stream a
  When :: (FinEnum a) ⇒ Stream a
            → (Stream b, b) → Stream b
  Merge :: (FinEnum a) ⇒ Stream a
            → Vec (Stream b) (Size a) → Stream b
    -- plus primitive operators

type Var a = String
class (Bounded a, Enum a) ⇒ FinEnum a where
  type Size a :: Nat
```

Fig. 3. Types used to implement *Haski*

in this chapter. Instead, we focus on the implementation details of the first phase, which also forms the basis for the IFC enforcement.

4 Haski as a Lustre Embedding

During the building pass, each line of a Haski program written using one of the combinators builds a corresponding intermediate *definition* under the hood of the *Haski* monad. These definitions are then parsed to construct a complete Lustre program in the node parsing pass. The purpose of this section is to describe the implementation of the building pass, and outline the action performed by the node parsing pass.

4.1 Building Recursive Definitions

The streams defined in the *Haski* monad are collected as a list of definitions. When run with an appropriate initial state, a Haski program produces a list of definitions which correspond to components of Lustre nodes. Definitions

are denoted by the *Def* data type, and expressions by *Stream* (see Figure 3). A definition may be a simple binding that binds a variable with a stream expression (*Let*), or an argument (*Arg*) or result (*Res*) of a node call.

The program *alt* from Section 2 builds the following definitions under the hood of the *Haski* monad.

Let "x" ((*Val Left*) ‘*Fby*‘ v_y)
Let "y" ((*Val Entered*) ‘*Fby*‘ v_x)

where v_x = *Var* "x" and v_y = *Var* "y". We use the same variables names as in the original program for readability, but this can also be implemented automatically with some compiler support [24].

Let us now turn to the implementation of combinators in the *Haski* monad. The combinator *fby* is implemented using the *letDef* combinator as follows.

```
fby :: HT a ⇒ a → Stream a → Haski (Stream a)
fby x s = letDef (Fby x s)
```

The combinator *letDef* is in turn implemented by adding a *Let* binding with a fresh variable name to the list of definitions in the *Haski* monad.

```
letDef :: Stream a → Haski (Stream a)
letDef s = do
  x ← freshVar
  addDef (Let x s)    -- updates state (‘defs‘)
  return (Var x)
```

It returns the variable in place of the original stream expression, thus replacing any use of the expression in later definitions with this variable. Returning a variable is the key to enabling recursive definitions without sending the Haski compiler into an infinite loop.

As *fby*, the implementation of *match* also builds definitions containing expressions under the hood, but is slightly more involved since *match* is derived from other expressions. We discuss this next.

4.2 Building Pattern Matching Definitions

The combinator *match* is overloaded in its function argument by the class *Streams* which has the following instances.

class *Streams b* **where**

  match :: (*FinEnum a*) ⇒ *Stream a* → (*a* → *b*) → *Haski b*

instance *Streams* (*Stream b*) **where** …

instance *Streams b* ⇒ *Streams* [*b*] **where** …

instance (*Streams b*, *Streams c*) ⇒ *Streams* (*b*, *c*) **where** …

  -- similarly for other "containers"

The overloading allows the *matching function* $a \to b$ to return multiple streams, such as lists or tuples of streams. In this section, we shall discuss the implementation of the instance *Streams* (*Stream b*). We skip the remaining instances since their implementation is mostly mechanical component-wise applications of *match*.

The combinator *match* provides a convenient interface for defining streams using the more fine-grained sampling operators *When* and *Merge*. For instance, the stream *resp* in the *cache* example from earlier defined using *match* on *req*, builds the following definition.

Let `"resp"` (v_{req} `` `Merge` `` [
  v_{state} `` `When` `` (v_{req}, *Read*)
  , (*Val Left*) `` `When` `` (v_{req}, *Write Left*)
  , (*Val Entered*) `` `When` `` (v_{req}, *Write Entered*)
])

When can be understood as a projection of a stream using another stream: the expression v_{state} `` `When` `` (v_{req}, *Read*) produces the value of v_{state} when the value of v_{req} is *Read*, and nothing otherwise. In the *Merge* expression above, the vector (written using list notation) contains a stream for each possible value of v_{req}. For every observed value of v_{req}, *Merge* produces the value of the corresponding stream in the vector. The use of *When* ensures that the branches of *Merge* are mutually complementary, which, as mentioned earlier in Section 2, is a restriction that is required of *Merge*.

Now consider implementing the instance *Match* (*Stream b*), where *match* has the type *FinEnum a* ⇒ *Stream a* → (*a* → *Stream b*) → *Haski* (*Stream b*). The matching function *a* → *Stream b* is expected to return an expression for every possible value of type *a*. To achieve the semantics of *match* illustrated above, we must implement *match* using *Merge*. But notice that *Merge*

requires a *vector* argument of type *Vec* (*Stream b*) (*Size a*) instead of a function, where *Size a* denotes the number of values that inhabit the type *a*. Using a vector forces a *Merge* expression to provide as many stream expressions as the number of values in the type *a* by construction, and thus enables the generated code to also inherit this property. This brings us to the question of implementing *match*: how must we construct a vector of streams from a function which returns them?

The solution to this problem is provided by the *FinEnum* class, which requires all its instances to be both bound and enumerable. Being bound and enumerable means that we could enumerate all the values of an instance type. Additionally, *FinEnum* is also finitely bound by the type family *Size*, which provides a type-level natural number of kind *Nat*. This enables us to enumerate the values as a vector of values, instead of a list of values. Let a function *enumerate* which does this be defined by the following class.

```
class FinEnum a ⇒ Enumerable a (n :: Nat) where
  enumerate :: Vec a n
```

Let us defer its implementation for the time being and simply assume that *enumerate* :: *Vec a* (*Size a*) returns all the values of type *a*.

Since the domain of the matching function is finitely enumerable, we can use *enumerate* to generate all possible arguments to the function. Moreover, we can also apply the function to the enumerated arguments to extract all possible results of the function. Thus we have a way to extract all the stream expressions returned by the function! This behavior is implemented by the following function—named after *"The Trick"* in partial evaluation [22].

```
theTrick :: FinEnum a ⇒ (a → b) → Vec b (Size a)
theTrick f = fmap f as
  where as :: Vec a (Size a) = enumerate
```

Equipped with *theTrick*, we implement the desired implementation of *match* as follows.

```
instance Streams (Stream b) where
  match s f = letDef $
  let body  = theTrick f
```

```
    whens = theTrick (λx → flip When (s, x))
  in Merge s (zipWith ($) whens body)
```

We first construct the vector which contains the streams on each branch of *Merge* in *body* :: *Vec* (*Stream b*) (*Size a*), and then insert the *When* expressions by zipping it (by application) with *whens* :: *Vec* (*Stream b* → *Stream b*) (*Size a*).

Recollect from earlier that the matching function is enforced to handle all the possible cases of its argument. We do not need any additional checks to enforce this behavior because this is already the case! If the function does not handle all possible cases, the invocation of the function *theTrick* by the compiler crashes with a `Non-exhaustive patterns` error—which, lucky for us, is exactly what we need!

It remains to implement *enumerate*, which is straightforward induction on the *Nat* parameter as follows[3].

```
instance Enumerable a 1 where
  enumerate = [ minBound ]
instance (Enum a, Enumerable a n, n'~n + 1)
  ⇒ Enumerable a n' where
  enumerate = succ (head ts) : ts
    where ts :: Vec a n = enumerate
```

The first value in the vector is constructed using *minBound* and the remaining elements are constructed by applying *succ* on the previous value. These functions are provided by the *Bounded* and *Enum* classes, respectively.

4.3 Building Nodes from Functions

As observed earlier, nodes are Haski subprograms that abstract over streams. Nodes are given a more liberal type which allows them to be regular Haskell functions that need not be defined inside the *Haski* monad. But this creates a challenge: how do we compile a Haskell function which represents a *Haski* node to a data representation of a Lustre node? Moreover, we cannot have a simple *Def* constructor that corresponds to a node call, since *Haski* nodes are not called with a special combinator.

[3] Requires UndecidableInstances and the OVERLAPPING pragma

To solve this problem, we first note that result of a node is always in the *Haski* monad. When fully applied, if we "register" each argument of a node call as a separate definition in the *Haski* monad, then we could recover the complete call in a later pass (node parsing) which acts on the collected list of definitions. The idea is to build definitions for a node when it is called, such that the definitions retain sufficient information for the node parsing pass to identify both *the node and its call*. For instance, we wish to build the following definitions for the call $prevAct \leftarrow cache\ (Val\ Entered)$.

Arg `"cache" "arg_1"` $(Val\ Entered)$
  Let `"resp"` $(v_{arg\_1}\ `Merge`\ [..])$
  Let `"state"` $((Val\ Left)\ `Fby`\ v_{resp})$
Res `"cache" "prevAct"` v_{resp}

The body of the *cache* node (containing *Let* definitions) is inlined at the call site by substituting its argument with a fresh variable (v_{arg_1}) instead of the actual argument *Val Entered*. From this invocation, we may recover both the body of the *cache* node and its invocation which defines *prevAct*—which is precisely the job of the node parsing pass. Multiple invocations of a node cause its body to be inlined multiple times, but the parsing pass simply ignores them if a node with a specific name has already been encountered.

Since functions may be partially applied, the arguments must be registered as they are received. Moreover, once all the arguments have been provided the resulting stream must be registered as one resulting from a node call. To achieve this, we shall wrap the function used to create a node inside another function which has the same type, but is also equipped with the ability to register the arguments and the result. This sneaky behavior is implemented by the *node* combinator.

The functions *argDef* and *resDef* (see Figure 4) provide an interface for registering arguments and result of a node. The instances *Arg* (*Stream a*) and *Res* (*Stream a*) allow a stream to be registered as an argument or a result respectively. Their implementation is similar to *letDef*. Additionally, a pair of arguments can also be registered by applying *argDef* on both components of the pair. As we shall see shortly, this instance has to do with registering multiple arguments.

```
class Arg a where
  argDef :: String → a → Haski a
class Res a where
  resDef :: String → a → Haski a
instance Arg (Stream a) where ...
instance (Arg a, Arg b) ⇒ Arg (a, b) where ...
instance Res (Stream a) where ...
```

Fig. 4. Interface used to register a node call

The combinator *node* is implemented by "boxing" the given function using a class *Box* which is overloaded in the return type of the function. It has two instances, *Box* (*Haski b*) for the base case where the function receives a single argument, and *Box* ($b \to c$) for the inductive case where the function receives more than one argument.

```
class Box b where
  node :: Arg a ⇒ String → (a → b) → (a → b)
instance (Res b) ⇒ Box (Haski b) where
  node name f = λe → do
    x' ← argDef name e
    r ← f x'
    r' ← resDef name r
    return r'
instance (Arg b, Box c) ⇒ Box (b → c) where
  node name f = curry (node name (uncurry f))
```

In the base case instance *Box* (*Haski b*), the function f has the type $a \to$ *Haski b*. To box this function, we register the argument using *argDef* and call the function with the result of the registration. This substitutes the occurrences of the argument in the body of the function with the variable returned by *argDef*. Finally we register the result of the function using *resDef* and return the corresponding definition.

For the inductive case, observe that we need to box a function $f :: a \to (b \to c)$, and the instance declaration provides us instances of *Arg b* and *Box c* as

the induction hypotheses. Additionally, we are also given an instance *Arg a* by the declaration of the function *node*. The instances *Arg a* and *Arg b* yield an instance for *Arg* (a, b). Thus, using instances *Arg* (a, b) and *Box c*, we can box the function f by currying it, and then uncurrying back to return the desired result.

5 Information-Flow Control

Haskell is well-known for providing information-flow control (IFC) through security libraries. These libraries ensure that code written using their API does not reveal secrets to unauthorized parties. Many of the existing (monadic) security libraries (e.g., SecLib [32], LIO [36], MAC [31], and HLIO [10]) are designed for writing secure code. In this work, however, we consider a different scenario where *we would like to extend an existing DSL to provide IFC security while minimizing changes to existing code*. Following this goal leads us to the design of an IFC enforcement where security checks are performed at code-generation time rather than at runtime (like in LIO) or type-checking time (like in MAC). In this section, we give a brief overview of IFC and explain the design choices of our IFC enforcement for Haski.

5.1 Security Lattices

IFC policies enforced by Haski are specified by a security lattice [15], which defines a partial order between security levels (*labels*). These labels represent the sensitivity of program inputs and outputs and the *order* between them dictates which flows of information are allowed in a program. Concretely, we write $\ell_1 \sqsubseteq \ell_2$ if data at security level ℓ_1 can flow to data ℓ_2 according to the security lattice. For example, the classic two-point lattice $\mathscr{L} = (\{L, H\}, \sqsubseteq)$ classifies data as either *public* (L) or *secret* (H) and only prohibits sending secret inputs into public outputs, i.e., $H \not\sqsubseteq L$.

5.2 Enforcement Design

We design a *coarse-grained* IFC enforcement [30], where developers only provide label annotations to security-relevant streams—rather than labeling every stream in a program. A labeled stream of type *LStream* is implemented by associating a stream expression with its label as follows.

```
  -- Labeled streams
data LStream a
  -- Manipulation of labeled streams
labelOf   :: LStream a → Label
label     :: Label → Stream a → Haski (LStream a)
unlabel   :: LStream a → Haski (Stream a)
  -- Current label
getLabel  :: Haski a → Haski Label
  -- Label creep avoidance
toLabeled :: Haski (Stream a) → Haski (LStream a)
```

Fig. 5. IFC interface for Haski

data *LStream a* = *LStream* { *getLabel* :: *Label*, *getStr* :: *Stream a* }

The type *LStream* acts as an opaque container since its implementation is not exposed to the programmer. For instance, the labeled stream *LStream Halexa* (*val* 42) is a constant stream that is confidential to the smart house controller *Halexa* .

Figure 5 shows Haski's IFC interface, which provides primitives to manipulate labeled streams while avoiding information leakage. Function *labelOf* obtains the label associated with a labeled stream. To understand the rest of the primitives, we need to introduce the concept of a *floating label*.

Every line in the *Haski* monad is associated with a special label known as the floating label (denoted by ℓ_f), which "floats above" the label of any observed stream during program execution, and thus represents an upper-bound on the sensitivity of all the streams in scope. The floating label is tracked in the state of the *Haski* monad:

data *HaskiSt* = *HaskiSt* { *defs* :: [*Def*], ℓ_f :: *Label*, … }

In order to enforce IFC policies, Haski regulates the interaction between *Haski* programs and labeled streams. *Haski* programs cannot write and read labeled streams directly, but must use the primitives in Figure 5. Let us discuss the implementation of these primitives next.

5.3 Implementing Labeling Primitives

The labeling primitives create and read labeled streams in compliance with specific security rules to avoid information leakage [4].

The primitive *label* labels a stream with a given label and does not affect the floating label of the program. Its implementation ensures that a desired label ℓ is at least the floating label of the program, i.e., $\ell_f \sqsubseteq \ell$, thus enforcing a *no write-down* policy. Intuitively, *label* creates a labeled stream as long as the decision to do so depends on less sensitive data. For example, given $\ell_f = L$, the invocation *label* H *s* (for some *s* :: *Stream Int*, for instance) is legal since $\ell_f \sqsubseteq H$. This means that a program which has read sensitive data cannot write public information in an attempt to leak it. If this criteria is not met, *label* inserts an error using *fail* in the *Haski* monad, thus crashing compilation.

The primitive *unlabel* acts as the dual of label and extracts the stream underlying a labeled stream. Unlike *label*, however, *unlabel* never crashes compilation and always succeeds. Instead, an invocation of *unlabel* on a stream s_l with label ℓ raises the floating label of the program to $\ell_f \sqcup \ell$.

Haski, as any other floating-label based IFC systems, suffers from the *label creep* problem. Unlabeling sensitive streams raises the floating label of the program, and hence a program which reads many sensitive streams risks raising its level to a point where it may not be able to produce any observable result. This problem is remedied using the *toLabeled* primitive, which addresses it by (i) creating a separate context where some sensitive computation can take place and (ii) restoring the original floating label afterwards.

The argument of *toLabeled* is a sensitive computation of type *Haski* (*Stream a*), that cannot return its result to the outer context—since that would be a leak. Instead, *toLabeled* wraps the result in a labeled stream using the floating label resulting from the execution of the sensitive computation. Unlike *unlabel*, *toLabeled* produces a labeled stream of type *LStream a* and its invocation does not affect the floating label. An invocation of *toLabeled* never crashes compilation.

5.4 Running Programs Securely

DC-labels Haski uses DC-labels [35], which is an expressive label format that can capture the security concerns of principals. DC-labels are pairs of

$$<C_1, I_1> \sqsubseteq <C_2, I_2> \iff (C_2 \Rightarrow C_1) \wedge (I_1 \Rightarrow I_2)$$

$$<C_1, I_1> \sqcup <C_2, I_2> \iff <C_1 \wedge C_2, I_1 \vee I_2>$$

$$<C_1, I_1> \sqcap <C_2, I_2> \iff <C_1 \vee C_2, I_1 \wedge I_2> \qquad \bot \equiv <True, False>$$

$$\top \equiv <False, True>$$

Fig. 6. DC-labels semantics

confidentiality and integrity policies, noted $<C, I>$ where C is the confidentiality policy and I is the integrity one. Both policies are positive propositional formulas in conjunctive normal form (CNF), where propositional constants represent *principals*. We assume that operations on formulas always reduce their results to CNF. For simplicity, we focus on confidentiality since the integrity part comes as a dual of it. Given two confidentiality policies C_1 and C_2, we interpret $<C_1, I> \sqsubseteq <C_2, I>$ as: C_2 is at least as confidential as C_1. For instance, $<Halexa \vee \text{Octavius}, I> \sqsubseteq <\text{Octavius}, I>$, which means that data readable by either *Halexa* or the Octavius is less confidential than data readable only by the Octavius. In contrast, given two integrity policies I_1 and I_2, we interpret $<C, I_1> \sqsubseteq <C, I_2>$ as: I_1 is more trustworthy than I_2, i.e., there are more principals taking responsibility for the data labeled with I_1 than with I_2. For instance, $<C, \text{Octavius} \wedge Halexa> \sqsubseteq <C, Halexa>$, which means that *Halexa* and the Octavius are jointly responsible for the data, which is more trustworthy than data only vouched by Octavius. Figure 6 presents the formalization of operations we will use in the rest of this section together with the definition of $\sqcup$ and $\sqcap$ in the security lattice. With DC-labels in place, we can associate the different components of our system to different principals, thus enabling them to impose different restrictions on the confidentiality and integrity of data.

Configuring security policies A Haski program that returns a stream (labeled or not) can be run using the *runAs* function on behalf of a principal. This function is intended to be used by an administrator who compiles a Haski program and assigns the right privilege to it—we assume that the administrator is part of the trusted computing base. Function *runAs* is defined as follows:

```
class IsStream f where
  runAs :: Haski (f a) → Principal → Haski (Label, Stream a)
```

The result of the *Haski* ($f\ a$) argument is overloaded in f to allow for both labeled and unlabeled streams to be returned. The *Principal* argument is used to set the initial floating label of the *Haski* computation and denotes the source of authority, i.e., the entity, that this program represents. For example, *runAs prog* `"Halexa"` runs a computation on behalf of Halexa with the DC-label <*Halexa*, *Halexa*>. As a result, any stream that is labeled by *prog* will contain *Halexa* in both the confidentiality and integrity components of its label—which means that the stream is confidential to *Halexa*, and also that *Halexa* has contributed to its content.

The *runAs* function returns a label that corresponds to the final floating label of the computation joined with the label of its result, along with the result that it returns. The returned label is intended to be used by the administrator to enforce application-specific security policies. Observe that the result is an unlabeled stream. This is due to the fact that *runAs* is run by the administrator, i.e., a person that we trust, so there is no need to protect the resulting stream by labeling it.

We implement the *runAs* function using the *toLabeled* primitive. This is because *toLabeled* allows us to create a separate context for the program to be run in, and restore the floating label of the administrator prior to execution. Restoring the floating label of the administrator allows the administrator to run programs on behalf of various principals without getting tainted by them. Here is the *Stream* instance which implements *runAs* for computations that return expressions.

```
instance IsStream Stream where
  runAs prog princ = do
    (LStream ℓ res) ← toLabeled $ do
      setLabel (newDCLabel princ princ)
      prog
    return (ℓ, res)
```

Function *setLabel* can only be used by the administrator and it is part of the trusted computed base, i.e., it is present in the IFC interface exposed to

```
type Status = Maybe Action
data WindowOp = Skip | Open | Close

halexa :: Stream Int → LStream Status
          → Haski (LStream WindowOp)
halexa = node "halexa" $ λtemp stat_l → do
  isHot ← letDef $ temp `gtE` 30
  toLabeled $ do
    stat ← unlabel stat_l
    pastAct ← (stat `match` mkReq) >>= cache
    recentAct ← stat `match` (maybe pastAct val)
    dec ← recentAct `match` λcase
      Left → val Close
      Entered → ifte isHot (val Open) (val Skip)
    return dec
  where
    mkReq :: Status → Stream Req
    mkReq Nothing = val Read
    mkReq (Just x) = val (Write x)
```

Fig. 7. Implementation of *Halexa*

developers. The function *newDCLabel* creates a label from the given principal by using it for both the confidentiality and integrity components.

The instance for the case of labeled expressions is implemented in turn using the above instance by simply unlabeling the result.

```
instance IsStream LStream where
  runAs prog princ = runAs (prog >>= unlabel) princ
```

The intended effect of this implementation is for the resulting label to be $\ell \sqcup \ell_f$, where ℓ_f is the floating label of *prog* at the end of its execution, and ℓ is the label of its result.

6 A Sample Application

In this section we illustrate the structure of the *Halexa* application and its security policy in Haski. The purpose of our application is to make a decision on opening a window, based on the current temperature in the house and

the status of the user Octavius. *Halexa* is expected to open the window when the temperature in the room is over 30°C provided Octavius is at home. If Octavius is not home, however, *Halexa* must close the window regardless of the temperature. We consider the status of Octavius sensitive information and thus require *Halexa* to confine the status and any information derived from it. That is, the status should not be used to build streams less sensitive than the DC-label <Octavius, Octavius>.

We model *Halexa* as a node which accepts two streams as arguments (see Figure 7): one of type *Stream Int* for the temperature reading, and another of type *LStream Status* for a labeled stream of notifications which notify *Halexa* about the actions of Octavius. The notifications specify whether Octavius has left (*Just Left*), entered (*Just Entered*), or that there is no change in status (*Nothing*). In response, the node returns a stream of instructions denoted by *Stream WindowOp* which instructs whether the window should be opened (*Open*), closed (*Close*), or whether nothing should be done (*Skip*). In essence, we implement *Halexa* using the *toLabeled* primitive to unlabel the labeled stream $stat_l$, thus ensuring that *Halexa* does not read its contents.

To understand the logic of the implementation, notice that a status stream *stat* need not contain any update in Octavius's action since it may be *Nothing*. Hence it is up to us to compute the whereabouts of Octavius from the *most recently observed action*. We compute this in the stream *recentAct* as follows: if the current value of *stat* is *Nothing* then use the last available action of the user (given by *pastAct*), else simply use the action given by *stat*. The stream *pastAct* retains the last action of the user using the *cache* node from earlier. Finally, we define a decision stream by matching on the *recentAct* stream, which produces the desired result. The combinator *ifte* is simply a shortened version of a *match* expression which pattern matches on *True* and *False*.

An administrator who wishes to run *Halexa* must provide the appropriate input streams to the node and assign the right policies using the function *runAs*. One such implementation is the following.

```
admin :: Haski (Stream WindowOp)
admin = do
  temp ← ...
  status ← ...
```

```
  status_l ← label ℓ_o status
  (res, ℓ) ← runAs (halexa temp status_l) (princ "Halexa")
  unless (ℓ ⊑ (ℓ_o ⊔ ℓ_h)) (fail "Bad Halexa")
  return res
  where
    ℓ_o = newDCLabel "Oct" "Oct"
    ℓ_h = newDCLabel "Halexa" "Halexa"
```

The security policy *unless…* in *admin* asserts that the resulting label must at most be a combination (⊔) of the labels of *Octavius* and *Halexa*. A simple case of obtaining the inputs would be to simply use fresh variables to define streams *temp* and *status*, which are then later initiated by the runtime. For a more realistic system, however, we require a way to obtain streams from entities outside of a Haski program. We discuss one possibility to address this requirement via bluetooth in the next section.

7 Reacting to Streams Outside of Haski

A typical IoT application communicates with several other applications and reacts to triggers which may originate from remote devices. To use Haski to build more realistic applications, it is important to enable streams to be provided by external sources. In this section, we consider the case of obtaining streams from remote devices via Bluetooth, which is a common means of communication in low power IoT devices. We manage to run *Halexa* by creating a small C runtime around the code generated by Haski. In essence, the runtime obtains the *temp* and *status* streams from earlier via the Bluetooth Low-Energy (BLE) API of Zephyr OS on the nrf52840DK board using the techniques discussed here with some manual intervention.

7.1 Briefly about Bluetooth Low Energy

The Bluetooth component we target uses the BLE stack on Zephyr OS[4], where the most common way that data flows through a BLE application is through a *Generic Attribute Profile* (GATT) server. Specifically, a device that has some data it wishes to make available to other devices will take the role of a GATT

[4]https://www.zephyrproject.org/

server. It will organise the data it has as *characteristics* that belong to *services*. As an example, a device might expose a biometrics service which in turn exposes the heart rate characteristic and the temperature characteristic.

A remote device that wishes to access or modify these values will take the role of a GATT client. A GATT client will initiate a connection to a GATT server, after which it scans for services and characteristics. Depending on the server configuration the client can update a remote characteristic, read a characteristic or subscribe to be notified about changes to a characteristic.

7.2 Preparing Halexa for Foreign Streams

A Haski program works on streams, yet the APIs we want to use in Zephyr OS use commands and callback functions. These need to be connected somehow.

For example, the Bluetooth API contains a function called *bt_gatt_subscribe* that is used to register a callback function whenever a message is received from a specified device. In Haski, when we subscribe to a device, we do not provide a callback function, but we receive a Haski stream instead:

$$btGattSubscribe :: DeviceID \longrightarrow Haski\ (Stream\ a)$$

So, for example, in order to connect the *Halexa* example from the previous section to the devices *tempSensor* and *motionSensor*, we can write the following code:

$$\begin{aligned} temp &\leftarrow btGattSubscribe\ tempSensor \\ status &\leftarrow btGattSubscribe\ motionSensor \\ &\ldots \end{aligned}$$

The compilation process will then generate an invocation of the C function *bt_gatt_subscribe* in the generated code and registers a callback to the *step* function—which is generated for every node—of *Halexa*. This means that the *step* function is called every time the devices *tempSensor* and *motionSensor* provide an update. Since the *step* function receives two arguments and the devices only produce one of them at a time, the *step* function is called with a default argument for the other. For example, the value of the *status* stream is *Nothing* when *tempSensor* provides an update.

7.3 The Halexa GATT Client

The BLE code that ties together the *Halexa* example with the remote temperature and the motion sensor assumes the role of a GATT client. The GATT client will scan for remote devices by calling the *bt_le_scan_start* BLE API function. The following function signatures have been simplified and rewritten in Haskell notation, and many less interesting functions have been omitted. The actual C versions of the API functions can be found in Appendix A.1.

bt_le_scan_start :: *ScanParams*
→ (*RemoteDeviceInfo* → *Int*) → *Int*

The second argument is a function that will be invoked when a device has been found. Once a remote device is found, a connection will be initiated with *bt_conn_le_create*.

bt_conn_le_create :: *RemoteAddress*
→ *CreateParams* → *ConnectionParams*
→ *Connection* → *Int*

When the connection has been established, we will scan it for the services it exposes. We expect to discover, e.g., the temperature service. To do this, we need to create some discovery parameters and then invoke *bt_gatt_discover*.

bt_gatt_discover :: *Connection* → *DiscoverParams* → *Int*

A subexpression of *DiscoverParams* is a function that will be called when a service have been discovered. This function will subscribe to a found service by invoking *bt_gatt_subscribe*. This will make sure that *Halexa* is notified about any changes to the remote temperature value.

bt_gatt_subscribe :: *Connection* → *SubscribeParams* → *Int*

The *SubscribeParams* contain a function that will be called every time a notification is received. The function will be invoked with values describing the connection that issued the notification as well as the actual payload.

Recollect from earlier that a node in Haski is compiled to *step* function in C which is invoked in response to the availability of its arguments. Compiling *Halexa* from the previous section generates a corresponding step function *halexa_step*. This function has the following signature.

```
Enum halexa_step (struct halexa_mem * self,
                  int temp, Enum motion)
```

In addition to this function, compiling *Halexa* also generates a struct *halexa_mem*, an instance of which is provided as the argument *self* to function *halexa_step*. This argument maintains the internal state of the stream returned by *Halexa*.

```
struct halexa_mem {...};
```

For every call of a node in a Haski program, an instance of such a struct is initialized globally before the first invocation, and passed as an argument to every subsequent invocation of the correponding *step* function. For *Halexa*, initialization is done as follows.

```
/* Global definition */
struct halexa_mem * mem;
...
/* Evaluated by main */
mem = k_malloc (sizeof (struct halexa_mem));
```

Using these definitions we build a function that is registered as a callback to be invoked whenever the BLE application receives, for example, a new temperature reading (as shown below).

```
static u8_t notify_temperature (..., const void * data ) {
    ...
    int * temperature = (int*) data ;
    ...
    halexa_step (mem, *temperature, NOTHING);
    ...
}
```

We invoke the function *halexa_step* with its internal memory *mem*, which stores the internal state of the node. Notice that we pass *NOTHING*, a representation of the corresponding Haskell value, for the status stream here. This is because the function *notify_temperature* is invoked in response to the temperature sensor, which does not provide a status update. A similar callback

function must be registered for the *status* stream by invoking *halexa_step* with a default temperature reading.

We emphasize that the small C runtime we implemented here is tailored to BLE and it requires some manual intervention to make the coupling between the generated code by Haski and Zephyr OS's API—we leave as future work to devise an automatic mechanism to do that.

7.4 Going Forward

The attentive reader might have paused to think while reading the previous section. The previous section describes how we compile a synchronous programming language to a target which uses callbacks and events instead of streams. It is not immediately obvious how to do this automatically. This discrepancy leads to the need for manual intervention when connecting the generated code to the outside world via BLE.

There are a few questions that need to be addressed in future work to bridge this gap. How is a continuous stream created from the sporadic events given to a callback function by the outside world? How do you compile a Haski node and dynamically register and unregister it as a callback?

We believe nicely generalising this is possible, and leave this and more questions as future work.

8 Related Work

Synchronous languages The seminal work of Lustre [12] (sometimes called "classical Lustre") shows how a declarative synchronous programming style can benefit from memory and computational time bounds. Lustre's ideas have been applied in a wide-range of scenarios ranging from hardware design (e.g., [7]) to real-time reactive systems (e.g., [29]).

Haski is based on a variation of classical Lustre from Biernacki et al. [6], the semantics of which has been formalized and verified by Auger et al. [1] and Bourke et al. [8]. The main difference between classical Lustre and the variant used by Haski is the absence of the *current* operator and the addition of the *merge* and *reset* operators. For a more detailed discussion on the differences, see Bourke et al. [8]. Haski does not (yet) implement the *reset* operator.

A notable implementation of Lustre that is closely related to ours is Lucid Synchrone [11]. Lucid Synchrone uses OCaml as the host language and allows a rich programming interface with many higher-order features of OCaml. Unlike Haski, it allows pattern matching on complex data types (e.g., streams of functions) that are not limited to finitely enumerable types. Naturally, the richer features offered by Lucid Synchrone also place higher demands from the runtime system, such as the need for a garbage collector. Haski, on the other hand, targets memory constrained IoT devices and thus strives to keep the runtime system minimal. The code generated by compiling a Haski program can be executed with a fixed amount of memory and does not require garbage collection.

Functional Reactive Programming Functional Reactive Programming (FRP) [16] is a programming style for programming asynchronous reactive systems. Unlike Lustre, it has the convenience of incorporating higher-order functions at the price of possibly introducing memory leaks—as noticed and addressed in subsequent work (e.g., [3, 14, 38]). Haski does not support higher-order functions as first class values, but enables developers to utilize them to build first-order Lustre programs. The staged programming approach ensures that all higher-order functions are eliminated at compile time, thus removing the need to address space leaks which may be caused by them.

Code generation for C We are not the first ones to propose an eDSL in Haskell for generating memory safe C code. Closest to our work is Copilot [28], an eDSL for stream-based programming for avionics. While Copilot provides similar guarantees on the generated code w.r.t. constant space and execution time, Haski presents a different programming experience (e.g., a monadic interface) as well as IFC security features. Haskino [19] is an eDSL to write programs to be run in an Arduino board while supporting a light-weight notion of threads. Like Haski, Haskino deploys the generated C code into a custom made runtime. Feldspar [2] is a DSL for describing digital signal processing algorithms in Haskell and generate C code. Ivory [17] is an advanced DSL for writing memory-safe C code within Haskell. It uses a simple notion of memory regions and also provides access control security checks to restrict side-effects in the generated C-code.

Language-based security for IoT Pyronia [23] provides access control and IFC for embedded devices written in Python. Pyronia runs under a custom-made runtime responsible to perform system call interposition, call stack inspection, and memory protection. Such modification are required to ensure that Python, where by design data is public, can safely execute and interact with C programs. In contrast, Haskell provides good abstractions to deliver a pure language-based IFC solution [31, 32, 36], which enables Haski to not require special runtimes and run on commodity IoT OSes. SainT [13] delivers an static IFC analysis for commodity SmartThings apps. SainT builds an intermediate representation for Groovy (object-oriented) SmartThings programs, where IFC checks are carried out. SainT targets legacy code while Haski provides security by construction using a coarse-grained IFC approach. Hence, SainT needs to extend the semantics of Groovy commands to reason about IFC. Instead, Haski provides modular security types (*LStream*) and primitives (e.g., *label* and *unlabel*) atop of our synchronous language. Velox VM [37] provides a Scheme virtual machine for constrained devices. Every app run by the VM has an associated *access control* policy file, which is used to restrict apps from accessing sensitive data and resource usage. As future work, Haski could integrate resource usage control as done by Velox VM.

Haskell security libraries The closest Haskell IFC libraries to our approach are LIO [31], HLIO [10], and MAC [36]. Our approach to enforce IFC at compile-time leads us to a new design space, where our API is a simplified version of the LIO's one due to executing the analysis at compile-time. More specifically, LIO takes an extra parameter in *toLabeled* to avoid leakage via labels [9], which Haski does not suffer from due to its static (compile-time) approach. Compared with HLIO and MAC, Haski is static but does not rely on Haskell's type-system for security checks but rather on the Haski compiler. Generally speaking, Haski's IFC API is a static, simplified, version of LIO's API while not going all the way to HLIO or MAC—it is something in between.

9 Final Remarks

We have presented Haski, a Haskell eDSL for writing software in embedded devices. Haski generates C code with memory consumption guarantees as well as information-flow security thanks to the many program analyses realized

by the compiler. We showcase that Haski programs can be easily integrated with a realistic runtime like the BLE in Zephyr OS. We expect this work to be a foundation to build IoT applications that leverage, not only BLE, but most of the underlying embedded OS functionality while providing security properties. Furthermore, we leave as future work to adapt our eDSL to allow users to be "in the loop" when relaxing IFC restrictions, e.g., to enable opening windows when the user is not home or to allow sending occupancy information to a security monitor firm. The Haski core development[5] (excluding the BLE runtime) currently consists of about 2600 lines of Haskell code.

In Retrospect

A missing element in this proposal is that it's unclear whether the Lustre primitives implemented by the eDSL, or the security solution offered by the paper are sufficient and practical to program real world IoT applications. Much remains in the scope of future work.

A Appendix

A.1 BLE API in C

In Section 7, the signatures of functions in the BLE API of Zephyr OS were rendered as Haskell types for the sake of brevity. Below, we show the original signatures for these methods in C.

int bt_le_scan_start (*const struct bt_le_scan_param* ∗ *param*,
bt_le_scan_cb_t device_found)

int bt_conn_le_create (*const bt_addr_le_t* ∗ *peer*,
const struct bt_conn_le_create_param ∗ *create_param*,
const struct bt_le_conn_para ∗ *conn_param*,
struct bt_conn ∗ *conn*)

int bt_gatt_discover (*struct bt_conn* ∗ *conn*,
struct bt_gatt_discover_params ∗ *params*)

int bt_gatt_subscribe (*struct bt_conn* ∗ *conn*,
struct bt_gatt_subscribe_params ∗ *params*)

[5] https://github.com/OctopiChalmers/haski

References

[1] Cédric Auger, J. L. Colaco, Grégoire Hamon, and Marc Pouzet. 2012. A Formalization and Proof of a Modular Lustre Compiler.

[2] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. 2010. Feldspar: A domain specific language for digital signal processing algorithms. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. 169–178.

[3] Patrick Bahr, Christian Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: a fitch-style modal calculus for reactive programming without space leaks. *Proc. ACM Program. Lang.* 3, ICFP (2019).

[4] David E. Bell and L. La Padula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation.* Technical Report MTR-2997, Rev. 1. MITRE Corporation, Bedford, MA.

[5] Elisa Bertino and Nayeem Islam. 2017. Botnets and Internet of Things Security. *IEEE Computer* 50, 2 (2017), 76–79.

[6] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. 2008. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*. 121–130.

[7] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*.

[8] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A formally verified compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 586–601.

[9] Pablo Buiras, Deian Stefan, and Alejandro Russo. 2014. On Dynamic Flow-Sensitive Floating-Label Systems. In *IEEE Computer Security Foundations Symposium, CSF*. 65–79.

[10] P. Buiras, D. Vytiniotis, and A. Russo. 2015. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM.

[11] Paul Caspi, Grégoire Hamon, and Marc Pouzet. 2008. Synchronous functional programming: The lucid synchrone experiment. *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes* (2008), 28–41.

[12] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. Lustre: A Declarative Language for Programming Synchronous Systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*.

[13] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick D. McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *USENIX Security*.

[14] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa arcade. In *Proc. of the ACM SIGPLAN Workshop on Haskell*. 7–18.

[15] D. E. Denning and P. J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977), 504–513.

[16] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*. 263–273.

[17] Trevor Elliott, Lee Pike, Simon Winwood, Patrick C. Hickey, James Bielman, Jamey Sharp, Eric L. Seidel, and John Launchbury. 2015. Guilt free ivory. In *Proc. of the ACM SIGPLAN Symposium on Haskell*. 189–200.

[18] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security Analysis of Emerging Smart Home Applications. In *IEEE Symposium on Security and Privacy, SP*. 636–654.

[19] Mark Grebe and Andy Gill. 2016. Threading the Arduino with Haskell. In *Trends in Functional Programming - 17th International Conference, TFP*. 135–154.

[20] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[21] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. 2011. Data Representation Synthesis. In *Proc. ACM Conference on Programming Language Design and Implementation*.

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

[23] Marcela S. Melara, David H. Liu, and Michael J. Freedman. 2019. Pyronia: Redesigning Least Privilege and Isolation for the Age of IoT. *CoRR* abs/1903.01950 (2019). arXiv:1903.01950 http://arxiv.org/abs/1903.01950

[24] Agustin Mista and Alejandro Russo. 2020. BinderAnn: Automated Reification of Source Annotations for Monadic EDSLs. In *21st International Symposium on Trends in Functional Programming, TFP*.

[25] Shayan Najd and Simon Peyton Jones. 2017. Trees that Grow. *Journal of Universal Computer Science* 23, 1 (2017), 42–62.

[26] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming, ICFP*.

[27] Matthew Pickering, Gergo Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern synonyms. In *Proc. of the 9th International Symposium on Haskell, Haskell 2016*.

[28] Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. 2013. Copilot: monitoring embedded systems. *ISSE* 9, 4 (2013), 235–255.

[29] Jie Qian, Jing Liu, Xiang Chen, and Junfeng Sun. 2015. Modeling and Verification of Zone Controller: The SCADE Experience in China's Railway Systems. In *1st IEEE/ACM International Workshop on Complex Faults and Failures in Large Software Systems, COUFLESS 2015*. 48–54.

[30] Vineet Rajani and Deepak Garg. 2018. Types for information flow control: Labeling granularity and semantic models. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 233–246.

[31] A. Russo. 2015. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM.

[32] A. Russo, K. Claessen, and J. Hughes. 2008. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM.

[33] A. Sabelfeld and A. C. Myers. 2003. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19.

[34] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. 2018. Situational Access Control in the Internet of Things. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*.

[35] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. 2011. Disjunction Category Labels. In *Proc. of the Nordic Conference on Information Security Technology for Applications (NORDSEC '11)*. Springer-Verlag.

[36] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*.

[37] Nicolas Tsiftes and Thiemo Voigt. 2018. Velox VM: A safe execution environment for resource-constrained IoT applications. *J. Netw. Comput. Appl.* 118 (2018).

[38] Atze van der Ploeg and Koen Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow!. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming, ICFP*. 302–314.

[39] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl A. Gunter. 2018. Fear and Logging in the Internet of Things. In *25th Annual Network and Distributed System Security Symposium, NDSS*.

[40] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proc. of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM.