Be My Guest

Nachiappan V.

Guest

Host

Normalization

Compilation

BCC

Agda

Sec

Agda

Haski

Haskell

Part I

BCC

Agda

2017

"Simplicity is a blockchain programming language that is so simple, it fits on a t-shirt."

- Russel O'Connor, Blockstream

2018

Simplicity and Michelson

Philip Wadler

University of Edinburgh and IOHK

Simplicity

"Simplicity is still too simple"

Towards Adding Variety to Simplicity

Authors | Authors and affiliations

Nachiappan Valliappan, Solène Mirliaz, Elisabet Lobo Vesga, Alejandro Russo

"Despite [being] capable of expressing non-trivial contracts, it can be very cumbersome to actually write one using its minimal constructs."

Towards Adding Variety to Simplicity

5 Adding Functions to Simplicity

In this section, we extend the Simplicity core language with user-defined func-

$$loop :: Types\ a \Rightarrow Simpl\ a\ a \to SNat \to Simpl\ a\ a$$

new features!

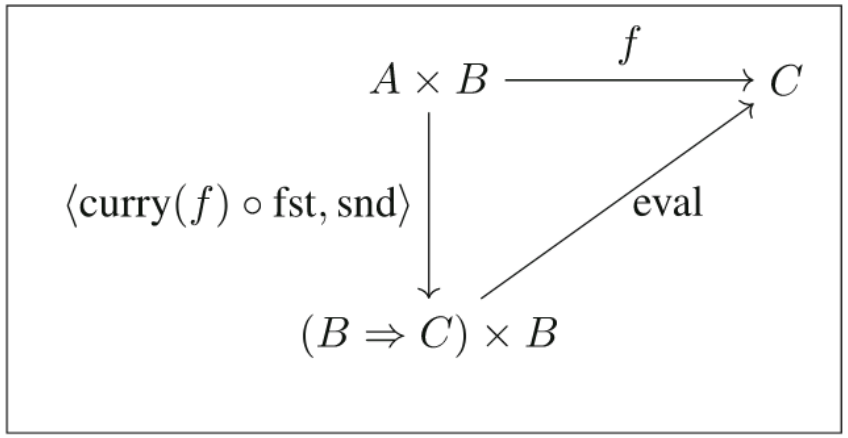


Fig. 9. Exponentials in BCCCs

categorical justification

Read Stack	Write Stack	List of exponentials
$[\ldots 1\underset{\uparrow}{1} \ldots]$	$[\ldots \underset{\uparrow}{?}???? \ldots]$	0000 $([NewFrame, Write\ 1, \ldots], [0010])$
$[\ldots]$	$[\ldots]$	

$\downarrow$ *run* $(PutClosure\ [Read, \ldots]\ 2\ 4)$

Read Stack	Write Stack	Closures list
$[\ldots 1\underset{\uparrow}{1} \ldots]$	$[\ldots\ 0001\underset{\uparrow}{?} \ldots]$	0000 $([NewFrame, Write\ 1, \ldots], [0010])$
$[\ldots]$	$[\ldots]$	0001 $([Read, \ldots], [11])$

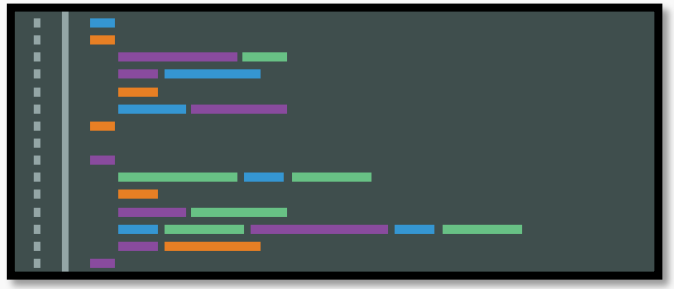
Fig. 12. Executing *PutClosure* in the SBM

Compiling to Categories

CONAL ELLIOTT, Target, USA

Some categories are cartesian but not *cartesian closed*, e.g., vector spaces with linear maps. Most of the rules for converting to CCC form rely on closure, which poses a problem for non-closed categories. If, however, the *overall* function being converted does not involve functions in its domain or codomain, then the corresponding closure-dependent CCC form can often (or perhaps always) be converted to a form free of the *Closed* operations (*apply*, *curry*, and *uncurry*)—assuming that none of the primitive operations (*addC*, *mulC*, *eq*, etc) involve exponentials in their types

Higher-order program $\{\mathbb{b}, 1, *, 0, +, \Rightarrow\}$



(concise, expressive)

?

READ	WRITE
100101	110100
...	...

(simple, lean)

First-order machine $\{\mathbb{b}, 1, *, 0, +\}$

BCC: Basic combinators

$a \longrightarrow b$

$$\mathrm{id} : a \longrightarrow a$$

$$\frac{f : b \longrightarrow c \qquad g : a \longrightarrow b}{f \circ g : a \longrightarrow c}$$

$$\frac{f : a \longrightarrow b \qquad g : a \longrightarrow c}{\mathrm{pair}\ f\ g : a \longrightarrow (b * c)}$$

$$\pi_1 : (a * b) \longrightarrow a \qquad \pi_2 : (a * b) \longrightarrow b$$

BCC: Function (or “exponential”) combinators

$$\frac{f : (a * b) \longrightarrow c}{\Lambda\ f : a \longrightarrow (b \Rightarrow c)}$$

$$\text{apply} : ((a \Rightarrow b) * a) \longrightarrow b$$

$$a \longrightarrow b \quad \xrightarrow{\ ?\ } \quad a \longrightarrow_1 b$$

$$\{\mathbb{b}, 1, *, 0, +, \Rightarrow\} \qquad \{\mathbb{b}, 1, *, 0, +\}$$

Yes!

Exponential Elimination for Bicartesian Closed Categorical Combinators

Nachiappan Valliappan
Chalmers University
Sweden
nacval@chalmers.se

Alejandro Russo
Chalmers University
Sweden
russo@chalmers.se

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

[Cousineau et al. 1987; Lafont 1988]. Abadi et al. [1991] 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

$$\text{and} : (\text{bool} * \text{bool}) \longrightarrow \text{bool}$$

$$\Lambda\ \text{and} : \text{bool} \to (\text{bool} \Rightarrow \text{bool})$$

$$\text{apply} \circ \text{pair}\ (\Lambda\ \text{and})\ \text{true} \xrightarrow{\text{norm}} \text{and} \circ (\text{pair}\ \text{true}\ \text{id})$$

$$: \text{bool} \to \text{bool} \qquad\qquad : \text{bool} \to \text{bool}$$

by rewriting?

Rewriting is difficult!

$$\{\mathbb{b}, 1, *, 0, +, \Rightarrow\}$$

No well-understood rewriting algorithms for combinators

Rewriting techniques for empty and sum types are daunting

Rewrite no more, piggyback!

Normalization by Evaluation

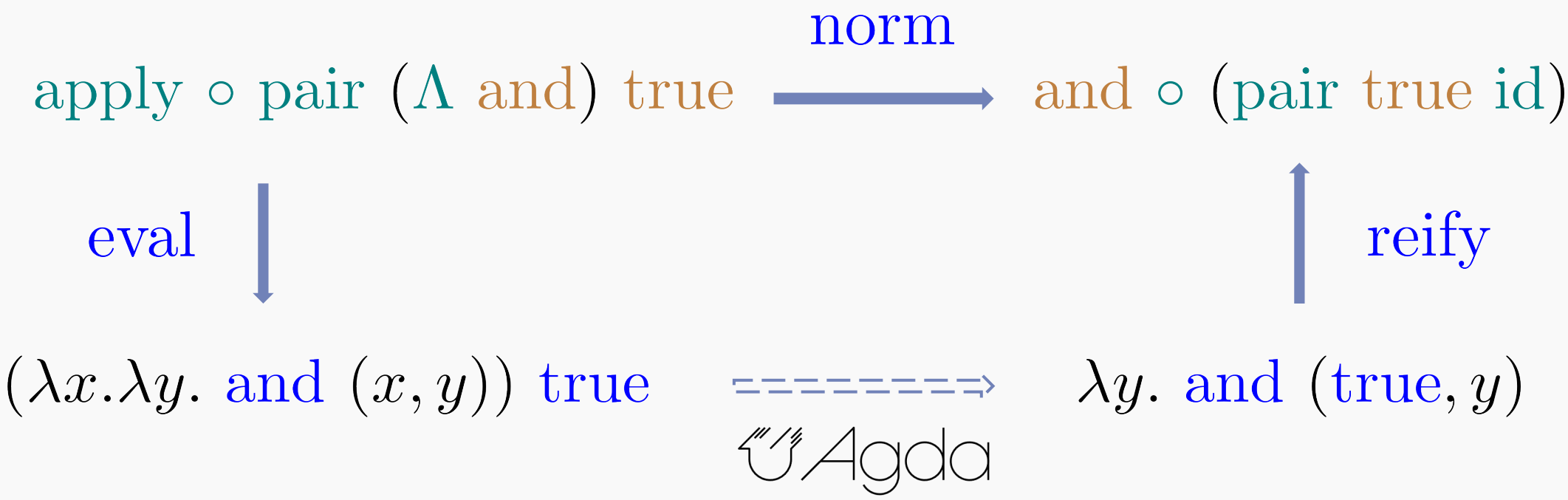
$$\mathrm{eval} : (a \longrightarrow b) \to (\llbracket a \rrbracket \dot{\longrightarrow} \llbracket b \rrbracket)$$

$$\mathrm{reify} : (\llbracket a \rrbracket \dot{\longrightarrow} \llbracket b \rrbracket) \to (a \longrightarrow_1 b)$$

$$\mathrm{norm} : (a \longrightarrow b) \to (a \longrightarrow_1 b)$$

$$\mathrm{norm}\ t = \mathrm{reify}\ (\mathrm{eval}\ t)$$

Normalization by Evaluation



Stories told and lessons learned

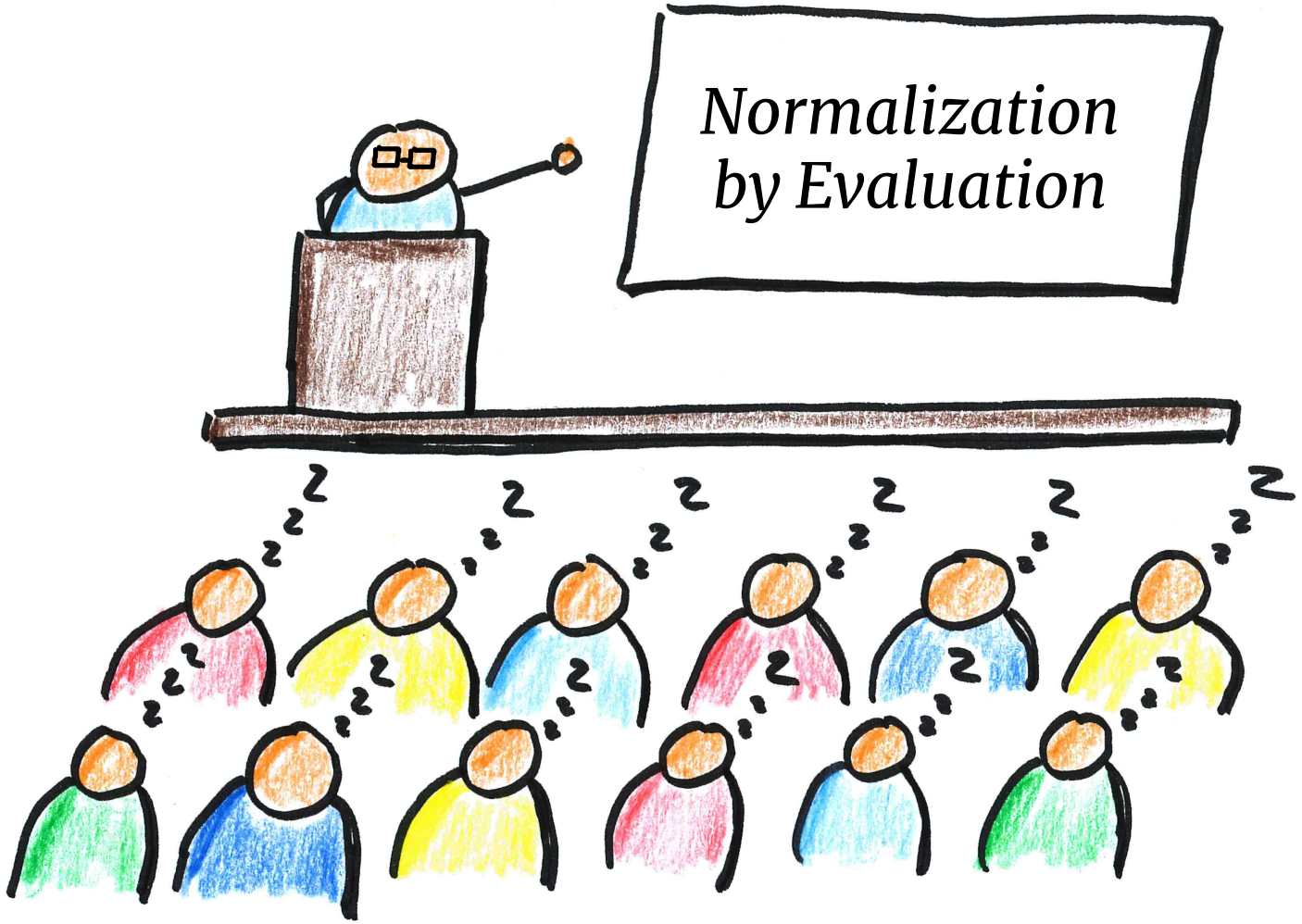
Proven correct, shown applicable

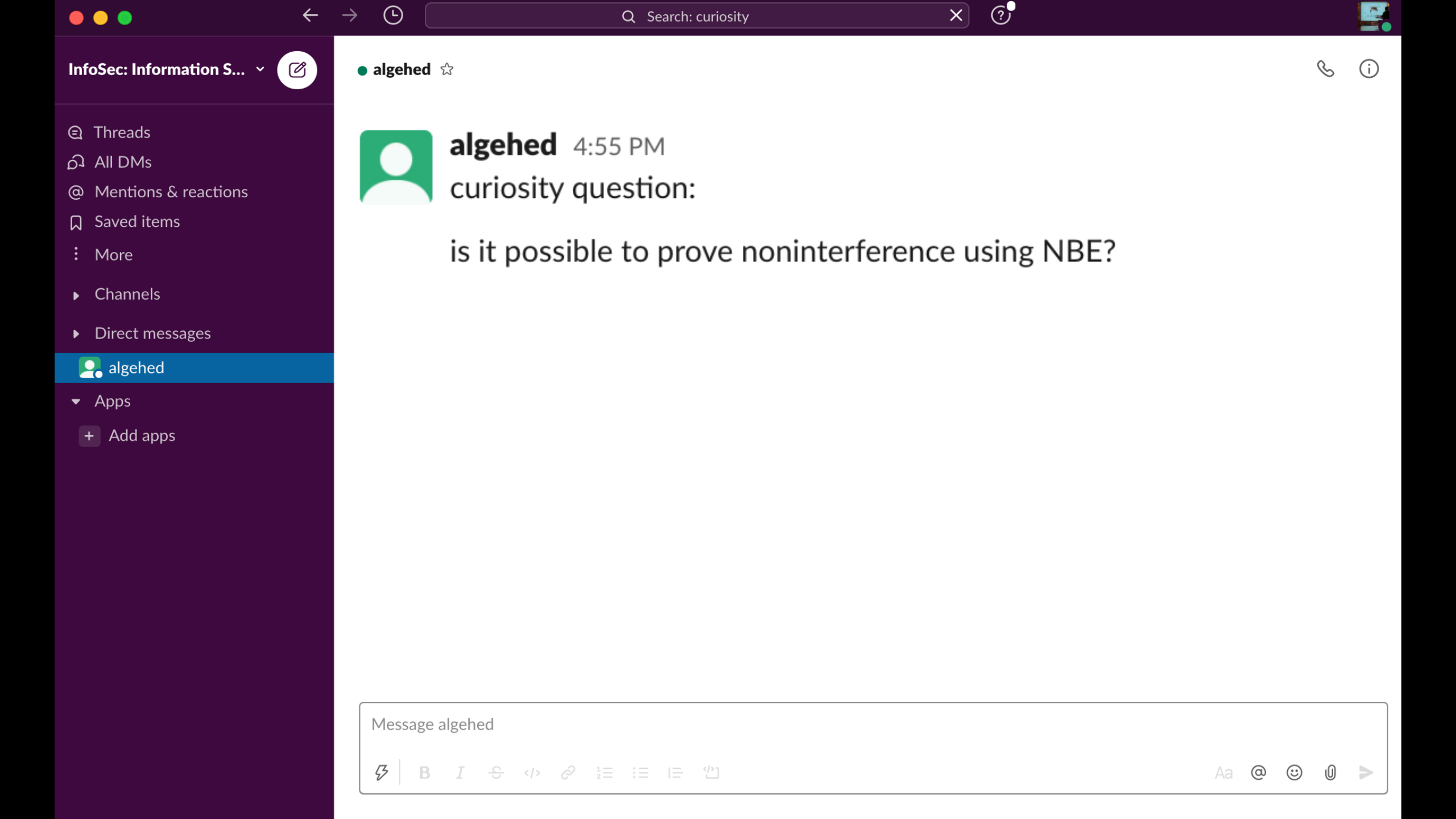
$$\mathrm{correct} : (f : a \longrightarrow b) \to \mathrm{norm}\ f \approx f$$

$$\mathrm{defunc} : \mathrm{Simpl}\ a\ b \to \mathrm{Simpl}_1\ a\ b$$

Insight: Target must be distributive!

$$\mathrm{distr} : a * (b + c) \longrightarrow_1 (a * b) + (a * c)$$



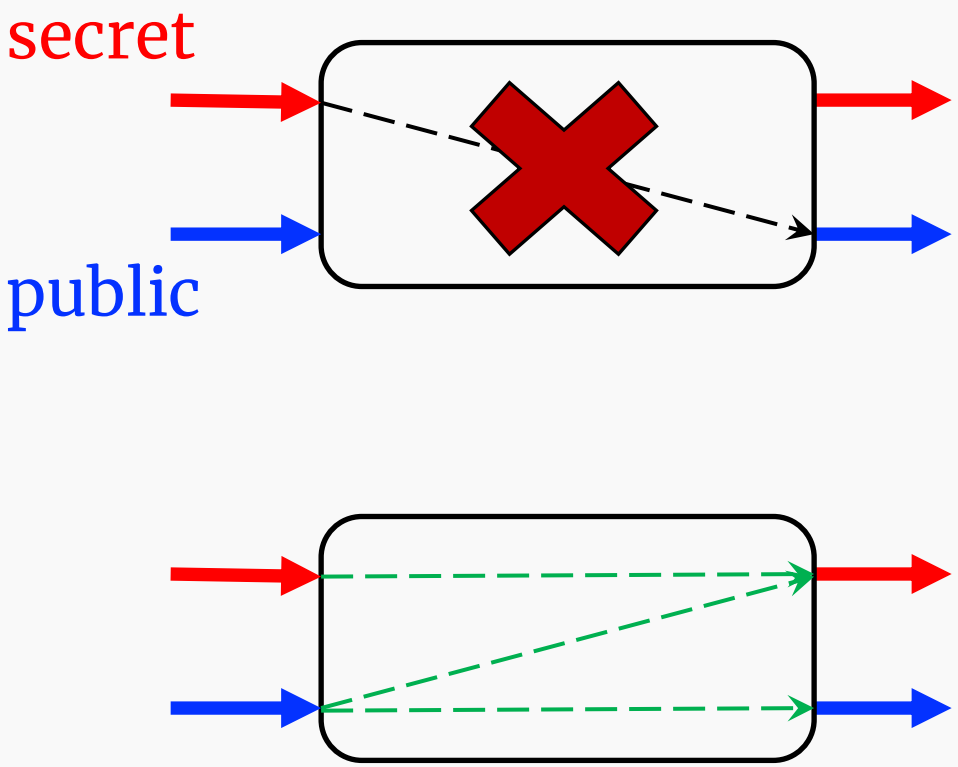

Search: curiosity
InfoSec: Information S...
Threads
All DMs
Mentions & reactions
Saved items
More
Channels
Direct messages
algehed
Apps
Add apps
algehed
algehed 4:55 PM
curiosity question:
is it possible to prove noninterference using NBE?
Message algehed

Part II

Sec

Agda

What is noninterference, anyway?



Sec: Model of seclib Hackage :: [Package]

$$\frac{\Gamma \vdash t : a}{\Gamma \vdash \mathrm{return}\ t : S\ \ell\ a}$$

label values

bind labeled values

$$\frac{\Gamma \vdash m : S\ \ell\ a \qquad \Gamma \vdash f : a \Rightarrow S\ \ell\ b}{\Gamma \vdash m \gg\!= f : S\ \ell\ b}$$

$$\frac{\Gamma \vdash m : S\ \ell\ a \qquad \ell \sqsubseteq \ell'}{\Gamma \vdash \mathrm{up}\ m : S\ \ell'\ a}$$

relabel values

How should you **prove** noninterference?

```
f : S H a -> S L bool
```

$f\ (sa_1) \rightsquigarrow \texttt{return true}$

$f\ (sa_2) \rightsquigarrow \texttt{return false}$

…

- Operational Semantics
- Denotational Semantics
- Parametricity
- *Normalization!*

What does normalization have to do with noninterference?

Inspect  `f : S H a -> S L bool`

- `f sa = return (not false)`
- `f sa = snd (sa >>= (\x . return x)`
  `, return false)`
- `f sa = …`

infinitely many arbitrarily complex possibilities…

Inspect  `f : S H a -> S L bool`

- `f sa = return (not false)`

  norm → `return true`

- `f sa = snd (sa >>= (\x . return x)`
  `, return false)`

  norm → `return false`

- `f sa = …`

  norm → `return true/false`

…that always normalize to a constant!

Idea: Normalize programs, and then inspect them.
If Sec guarantees noninterference, then programs of form secret → public must be constant.

Simple Noninterference by Normalization

Carlos Tomé Cortiñas*
Chalmers University of Technology
carlos.tome@chalmers.se

Nachiappan Valliappan*
Chalmers University of Technology
nacval@chalmers.se

Abstract

Information-flow control (IFC) languages ensure programs preserve the confidentiality of sensitive data. *Noninterference*, the desired security property of such languages, states

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.

Normalize how? By Evaluation!

$$\{\mathbb{b}, 1, *, 0, +, \Rightarrow, S\ \ell\}$$

reuse previous work!

$$\mathrm{eval} : (\Gamma \vdash a) \to (\llbracket\Gamma\rrbracket \dot{\longrightarrow} \llbracket a\rrbracket)$$

$$\mathrm{reify} : (\llbracket\Gamma\rrbracket \dot{\longrightarrow} \llbracket a\rrbracket) \to (\Gamma \vdash_{\mathrm{nf}} a)$$

$$\mathrm{norm} : (\Gamma \vdash a) \to (\Gamma \vdash_{\mathrm{nf}} a)$$

$$\mathrm{norm}\ t = \mathrm{reify}\ (\mathrm{eval}\ t)$$

Stories told and lessons learned

Proven correct and secure

$$\mathrm{correct} : (\Gamma \vdash t : a) \to \mathrm{norm}\ t \approx t$$

$$\mathrm{secure} : (\Gamma_\ell \vdash t : S\ \ell'\ \tau) \to \ell \sqsubseteq \ell' \ \vee\ (\mathrm{IsConst}\ t)$$

Insight: Type-safe noninterference can be proved syntactically!

Normalization isn’t the only opportunity, **compilation too**!

Part III

Haski

Haskell

Haski: DSL for programming streams based on Lustre

$val\ x$	*construct constant stream x, x, x, x,...*
$fby\ x\ s$	*construct a stream x, s_1, s_2, s_3,...*
$match\ s\ \{$	*pattern-match over s and return b_i for a_i*
$a_1 \rightarrow b_1$	
$\ldots$	
$a_n \rightarrow b_2\ \}$	

Embedding Haski in Haskell

```
val    :: a -> Haski (Stream a)
fby    :: a -> Stream a -> Haski (Stream a)
match :: … => Stream a -> (a -> Stream b)
              -> Haski (Stream b)
…
```

Free parsing and type-checking!

```
code_gen :: Haski (Stream a) -> Code
```

Compiling Recursive definitions

```
nats :: Haski (Stream Int)
nats = mdo
   x <- 0 `fby` x + 1
   return x

fib :: Haski (Stream Int)
fib = mdo
   x <- 0 `fby` y
   y <- 1 `fby` (x + y)
   return x
```

Compiling Pattern matching

```
data Loc = In | Out

flipper :: Stream Loc -> Haski (Stream Loc)
flipper loc = mdo
   loc' <- loc `match` \case
         In  -> Out
         Out -> In
    return loc'
```

switch loc {
   case In: loc' = Out
   case Out: loc' = In
}

Compiling Pattern matching using “The Trick”

```
code_gen' :: Stream a -> Code
```

Require type ‘a’ to be *finitely enumerable*

```
match :: FinEnum a
   => Stream a -> (a -> Stream b)
   -> Haski (Stream b)
```

Enumerate $[a_0, a_1, a_2, \ldots, a_n]$

map with `(a -> Stream b)`

Enabling Information-Flow Control (IFC)

Labeled Stream

```
sec_flipper :: LStream Loc -> Haski (LStream Loc)
```

IFC primitives

```
label (High :: Label) (loc :: Stream Loc)
      :: LStream Loc

unlabel (sloc :: LStream Loc)
      :: Haski (Stream Loc)
…
```

Programming with IFC primitives

```
sec_flipper :: LStream Loc -> Haski (LStream Loc)
sec_flipper sloc = mdo
   loc <- unlabel sloc
   loc' <- flipper loc
   sloc' <- label High loc'
    return sloc'
```

checked during compilation

Static IFC, but not using types!

- Labels are *static values*, not types

  i.e., `High :: Label`, not `High :: *`

- IFC primitives are "compiled away"

  i.e., `sec_flipper ≈ flipper`, if there's no violation

Using Haski for programming IoT devices

Towards Secure IoT Programming in Haskell

Nachiappan Valliappan
Chalmers University of Technology
Gothenburg, Sweden
nacval@chalmers.se

Robert Krook
Chalmers University of Technology
Gothenburg, Sweden
krookr@chalmers.se

Alejandro Russo
Chalmers University of Technology
Gothenburg, Sweden
russo@chalmers.se

Koen Claessen
Chalmers University of Technology
Gothenburg, Sweden
koen@chalmers.se

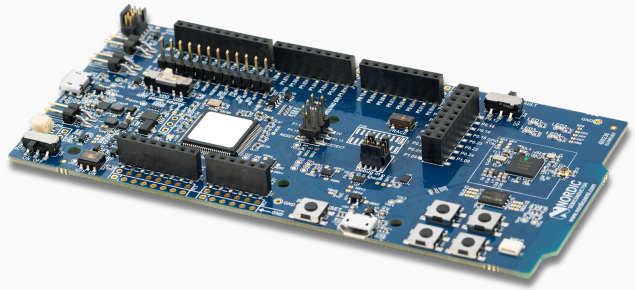
Abstract

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

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., [Bertino and Islam 2017; Fernandes et al. 2016; Schuster et al. 2018; Wang et al. 2018]). Hence, it has become important to develop security solutions which address the concerns of unauthorized access to data and privacy loss.

Stories told and lessons learned

Haski on Zephyr™ + 

(RTOS with a C API) (microcontroller)

Insight: eDSLs are amenable to use of partial evaluation techniques

Normalization

Compilation

BCC

Agda

Sec

Agda

Haski

Haskell