

# A Dependent Dependency Calculus

Pritam Choudhury  
Harley Eades III  
Stephanie Weirich

University of Edinburgh LFCS  
June 14, 2022

- Consider

$$x{:}^L \mathbf{Int}, y{:}^H \mathbf{Bool}, z{:}^M \mathbf{Bool} \vdash \text{if } z \text{ then } x \text{ else } 3 :^M \mathbf{Int}$$

where the type system is parameterized by a lattice ( $L < M < H$ )

- Consider

$$x:^L \mathbf{Int}, y:^H \mathbf{Bool}, z:^M \mathbf{Bool} \vdash \text{if } z \text{ then } x \text{ else } 3 :^M \mathbf{Int}$$

where the type system is parameterized by a lattice ( $L < M < H$ )

- Noninterference:** If  $x:\ell_1 A \vdash b:\ell_2 B$  and  $\ell_1 \not\leq \ell_2$  then  $b$  cannot *depend* on  $x$  during computation.

- Consider

$$x :^L \mathbf{Int}, y :^H \mathbf{Bool}, z :^M \mathbf{Bool} \vdash \text{if } z \text{ then } x \text{ else } 3 :^M \mathbf{Int}$$

where the type system is parameterized by a lattice ( $L < M < H$ )

- Noninterference:** If  $x :^{\ell_1} A \vdash b :^{\ell_2} B$  and  $\ell_1 \not\leq \ell_2$  then  $b$  cannot *depend* on  $x$  during computation.
- Applications:* Security types (information flow, provenance), Compiler optimizations (binding-time analysis), etc.  
Related to Dependency Core Calculus: Abadi et al. (1999), Sealing Calculus: Shikuma and Igarashi (2006)

# Goal: Irrelevance in Dependent Type Theories

- Generalize dependency analysis to *dependent type systems*

# Goal: Irrelevance in Dependent Type Theories

- Generalize dependency analysis to *dependent type systems*
- **Why?** Use dependency to track two forms of *irrelevance*
  - **Run-time irrelevance:** some parts of terms can be *erased* before execution
  - **Compile-type irrelevance:** some parts of terms can be *ignored* when checking type equivalence

# Dependency and simple types

$$\boxed{\Gamma \vdash a :^\ell A}$$

SDC-VAR

$$\frac{\ell_0 \leq \ell \quad x :^{\ell_0} A \in \Gamma}{\Gamma \vdash x :^\ell A}$$

SDC-ABS

$$\frac{\Gamma, x :^\ell A \vdash b :^\ell B}{\Gamma \vdash \lambda x : A. b :^\ell A \rightarrow B}$$

SDC-APP

$$\frac{\Gamma \vdash b :^\ell A \rightarrow B \quad \Gamma \vdash a :^\ell A}{\Gamma \vdash b a :^\ell B}$$

*(Simple types)*

# Dependency and simple types

$$\boxed{\Gamma \vdash a :^\ell A}$$

(Simple types)

SDC-VAR

$$\frac{\ell_0 \leq \ell \quad x :^{\ell_0} A \in \Gamma}{\Gamma \vdash x :^\ell A}$$

SDC-ABS

$$\frac{\Gamma, x :^\ell A \vdash b :^\ell B}{\Gamma \vdash \lambda x : A. b :^\ell A \rightarrow B}$$

SDC-APP

$$\frac{\Gamma \vdash b :^\ell A \rightarrow B \quad \Gamma \vdash a :^\ell A}{\Gamma \vdash b a :^\ell B}$$

Internalize judgment with graded modal type

$T^{\ell_0} A$  describes terms of type  $A$  checked at least at level  $\ell_0$



# Dependency and simple types

$$\boxed{\Gamma \vdash a :^\ell A}$$

(Simple types)

SDC-VAR

$$\frac{\ell_0 \leq \ell \quad x :^{\ell_0} A \in \Gamma}{\Gamma \vdash x :^\ell A}$$

SDC-ABS

$$\frac{\Gamma, x :^\ell A \vdash b :^\ell B}{\Gamma \vdash \lambda x : A. b :^\ell A \rightarrow B}$$

SDC-APP

$$\frac{\Gamma \vdash b :^\ell A \rightarrow B \quad \Gamma \vdash a :^\ell A}{\Gamma \vdash b a :^\ell B}$$

SDC-RETURN

$$\frac{\Gamma \vdash a :^{\ell \vee \ell_0} A}{\Gamma \vdash \eta^{\ell_0} a :^\ell T^{\ell_0} A}$$

SDC-BIND

$$\frac{\Gamma \vdash a :^\ell T^{\ell_0} A \quad \Gamma, x :^{\ell \vee \ell_0} A \vdash b :^\ell B}{\Gamma \vdash \mathbf{bind}^{\ell_0} x = a \mathbf{in} b :^\ell B}$$

# Dependency and simple types

$$\boxed{\Gamma \vdash a :^\ell A}$$

(Simple types)

SDC-VAR

$$\frac{\ell_0 \leq \ell \quad x :^{\ell_0} A \in \Gamma}{\Gamma \vdash x :^\ell A}$$

SDC-ABS

$$\frac{\Gamma, x :^\ell A \vdash b :^\ell B}{\Gamma \vdash \lambda x : A. b :^\ell A \rightarrow B}$$

SDC-APP

$$\frac{\Gamma \vdash b :^\ell A \rightarrow B \quad \Gamma \vdash a :^\ell A}{\Gamma \vdash b a :^\ell B}$$

SDC-RETURN

$$\frac{\Gamma \vdash a :^{\ell \vee \ell_0} A}{\Gamma \vdash \eta^{\ell_0} a :^\ell T^{\ell_0} A}$$

SDC-BIND

$$\frac{\Gamma \vdash a :^\ell T^{\ell_0} A \quad \Gamma, x :^{\ell \vee \ell_0} A \vdash b :^\ell B}{\Gamma \vdash \mathbf{bind}^{\ell_0} x = a \mathbf{in} b :^\ell B}$$

Equivalent elimination form:  $\mathbf{unseal}^{\ell_0} a \triangleq \mathbf{bind}^{\ell_0} x = a \mathbf{in} x$

# Dependency and simple types

$$\boxed{\Gamma \vdash a :^\ell A}$$

(Simple types)

SDC-VAR

$$\frac{\ell_0 \leq \ell \quad x :^{\ell_0} A \in \Gamma}{\Gamma \vdash x :^\ell A}$$

SDC-ABS

$$\frac{\Gamma, x :^\ell A \vdash b :^\ell B}{\Gamma \vdash \lambda x : A. b :^\ell A \rightarrow B}$$

SDC-APP

$$\frac{\Gamma \vdash b :^\ell A \rightarrow B \quad \Gamma \vdash a :^\ell A}{\Gamma \vdash b a :^\ell B}$$

SDC-BIND

$$\frac{\Gamma \vdash a :^\ell T^{\ell_0} A \quad \Gamma, x :^{\ell \vee \ell_0} A \vdash b :^\ell B}{\Gamma \vdash \mathbf{bind}^{\ell_0} x = a \mathbf{in} b :^\ell B}$$

SDC-RETURN

$$\frac{\Gamma \vdash a :^{\ell \vee \ell_0} A}{\Gamma \vdash \eta^{\ell_0} a :^\ell T^{\ell_0} A}$$

SEALING-UNSEAL

$$\frac{\Gamma \vdash a :^\ell T^{\ell_0} A \quad \ell_0 \leq \ell}{\Gamma \vdash \mathbf{unseal}^{\ell_0} a :^\ell A}$$

# Indexed indistinguishability

Define *indexed indistinguishability* as  $\boxed{\Phi \vdash a \sim_\ell b}$  when

- $a$  and  $b$  differ only in places marked by  $\eta^{\ell_0}$ , where  $\neg(\ell_0 \leq \ell)$ ,
- $a$  and  $b$  depend only on variables  $x: \ell_0 \in \Phi$ , where  $\ell_0 \leq \ell$ .

# Indexed indistinguishability

Define *indexed indistinguishability* as  $\boxed{\Phi \vdash a \sim_\ell b}$  when

- $a$  and  $b$  differ only in places marked by  $\eta^{\ell_0}$ , where  $\neg(\ell_0 \leq \ell)$ ,
- $a$  and  $b$  depend only on variables  $x: \ell_0 \in \Phi$ , where  $\ell_0 \leq \ell$ .

Public observers (at level  $L$ ) are oblivious to secret data (marked  $H$ ).

$$f: L \vdash f(\eta^H \mathbf{True}) \sim_L f(\eta^H \mathbf{False})$$

# Indexed indistinguishability

Define *indexed indistinguishability* as  $\boxed{\Phi \vdash a \sim_\ell b}$  when

- $a$  and  $b$  differ only in places marked by  $\eta^{\ell_0}$ , where  $\neg(\ell_0 \leq \ell)$ ,
- $a$  and  $b$  depend only on variables  $x: \ell_0 \in \Phi$ , where  $\ell_0 \leq \ell$ .

Public observers (at level  $L$ ) are oblivious to secret data (marked  $H$ ).

$$f: L \vdash f(\eta^H \mathbf{True}) \sim_L f(\eta^H \mathbf{False})$$

High-level observers can make more distinctions.

$$f: L \vdash f(\eta^H \mathbf{True}) \not\sim_H f(\eta^H \mathbf{False})$$

# Indexed indistinguishability

Define *indexed indistinguishability* as  $\boxed{\Phi \vdash a \sim_\ell b}$  when

- $a$  and  $b$  differ only in places marked by  $\eta^{\ell_0}$ , where  $\neg(\ell_0 \leq \ell)$ ,
- $a$  and  $b$  depend only on variables  $x: \ell_0 \in \Phi$ , where  $\ell_0 \leq \ell$ .

Public observers (at level  $L$ ) are oblivious to secret data (marked  $H$ ).

$$f: L \vdash f (\eta^H \mathbf{True}) \sim_L f (\eta^H \mathbf{False})$$

High-level observers can make more distinctions.

$$f: L \vdash f (\eta^H \mathbf{True}) \not\sim_H f (\eta^H \mathbf{False})$$

Indexed indistinguishability is an equivalence relation and closed under substitution.

# Syntactic proof of Noninterference

**Theorem (Operational semantics respects indexed indistinguishability)**

*If  $\Phi \vdash a_1 \sim_\ell a'_1$  and  $a_1 \rightsquigarrow a_2$  then there exists some  $a'_2$  such that  $a'_1 \rightsquigarrow a'_2$  and  $\Phi \vdash a_2 \sim_\ell a'_2$ .*



# Syntactic proof of Noninterference

## Theorem (Operational semantics respects indexed indistinguishability)

*If  $\Phi \vdash a_1 \sim_\ell a'_1$  and  $a_1 \rightsquigarrow a_2$  then there exists some  $a'_2$  such that  $a'_1 \rightsquigarrow a'_2$  and  $\Phi \vdash a_2 \sim_\ell a'_2$ .*

## Corollary

*Given  $x :^H A \vdash b :^L \mathbf{Int}$  and  $\emptyset \vdash a_1, a_2 :^H A$ , if  $\vdash b\{a_1/x\} \rightsquigarrow^* v_1$  and  $\vdash b\{a_2/x\} \rightsquigarrow^* v_2$  then  $v_1 = v_2$ .*

# Label-indexed definitional equality

Define *label-indexed definitional equality*,  $\Phi \vdash a \equiv_{\ell} b$  as the closure of indexed indistinguishability by  $\beta$ -reduction.

## Lemma (Substitution)

Given  $\Phi, x : \ell_0 \vdash b_1 \equiv_{\ell} b_2$ .

- 1 If  $\ell_0 \leq \ell$  and  $\Phi \vdash a_1 \equiv_{\ell} a_2$  then  $\Phi \vdash b_1\{a_1/x\} \equiv_{\ell} b_2\{a_2/x\}$ .
- 2 If  $\neg(\ell_0 \leq \ell)$  then  $\Phi \vdash b_1\{a_1/x\} \equiv_{\ell} b_2\{a_2/x\}$ .

# Dependent Dependency Calculus (DDC)

- DDC is a Pure Type System extended with an arbitrary lattice of dependency levels  $\ell$
- $\Pi$  and  $\Sigma$  types annotated with levels ( $\Pi x :^\ell A. B$  and  $\Sigma x :^\ell A. B$ )

# Dependent Dependency Calculus (DDC)

- DDC is a Pure Type System extended with an arbitrary lattice of dependency levels  $\ell$
- $\Pi$  and  $\Sigma$  types annotated with levels ( $\Pi x:^{\ell} A.B$  and  $\Sigma x:^{\ell} A.B$ )
- $\Sigma$  types encode labelled graded type:  $T^{\ell} A \triangleq \Sigma x:^{\ell} A.\mathbf{unit}$

# Dependent Dependency Calculus (DDC)

- DDC is a Pure Type System extended with an arbitrary lattice of dependency levels  $\ell$
- $\Pi$  and  $\Sigma$  types annotated with levels ( $\Pi x :^\ell A. B$  and  $\Sigma x :^\ell A. B$ )
- $\Sigma$  types encode labelled graded type:  $T^\ell A \triangleq \Sigma x :^\ell A. \mathbf{unit}$
- Lattice must have a distinguished element  $C$ , with  $\perp \leq C \leq \top$

# Dependent Dependency Calculus (DDC)

- DDC is a Pure Type System extended with an arbitrary lattice of dependency levels  $\ell$
- $\Pi$  and  $\Sigma$  types annotated with levels ( $\Pi x:\ell A.B$  and  $\Sigma x:\ell A.B$ )
- $\Sigma$  types encode labelled graded type:  $T^\ell A \triangleq \Sigma x:\ell A.\mathbf{unit}$
- Lattice must have a distinguished element  $C$ , with  $\perp \leq C \leq \top$
- Definitional equality:  $\Phi \vdash a \equiv_C b$

# Dependent Dependency Calculus (DDC)

- DDC is a Pure Type System extended with an arbitrary lattice of dependency levels  $\ell$
- $\Pi$  and  $\Sigma$  types annotated with levels ( $\Pi x:\ell A.B$  and  $\Sigma x:\ell A.B$ )
- $\Sigma$  types encode labelled graded type:  $T^\ell A \triangleq \Sigma x:\ell A.\mathbf{unit}$
- Lattice must have a distinguished element  $C$ , with  $\perp \leq C \leq \top$
- Definitional equality:  $\Phi \vdash a \equiv_C b$
- Dependency levels intuition
  - Executable:  $\Gamma \vdash a :^\perp A$
  - Comparable:  $\Gamma \vdash a :^C A$
  - Irrelevant:  $\Gamma \vdash a :^\top A$

# Dependent Dependency Calculus (DDC)

- DDC is a Pure Type System extended with an arbitrary lattice of dependency levels  $\ell$
- $\Pi$  and  $\Sigma$  types annotated with levels ( $\Pi x :^\ell A.B$  and  $\Sigma x :^\ell A.B$ )
- $\Sigma$  types encode labelled graded type:  $T^\ell A \triangleq \Sigma x :^\ell A.\mathbf{unit}$
- Lattice must have a distinguished element  $C$ , with  $\perp \leq C \leq \top$
- Definitional equality:  $\Phi \vdash a \equiv_C b$
- Dependency levels intuition
  - Executable:  $\Gamma \vdash a :^\perp A$
  - Comparable:  $\Gamma \vdash a :^C A$
  - Irrelevant:  $\Gamma \vdash a :^\top A$
- When  $\Gamma \vdash a :^\ell A$  must have  $\ell \leq C$   
(Define  $\Gamma \vdash a :^\top A$  using “resurrection”, i.e.  $C \wedge \Gamma \vdash a :^C A$ )



# Irrelevance for Dependent Types

Use lattice  $\perp < C < T$

- Dependency analysis for **run-time** irrelevance
  - What parts of the program can we safely *erase before execution*?

# Irrelevance for Dependent Types

Use lattice  $\perp < C < T$

- Dependency analysis for **run-time** irrelevance
  - What parts of the program can we safely *erase before execution*?
  - Type check all “executable” parts of the program at level  $\perp$  and “eraseable” parts *at least* at level  $C$

# Irrelevance for Dependent Types

Use lattice  $\perp < C < T$

- Dependency analysis for **run-time** irrelevance
  - What parts of the program can we safely *erase before execution*?
  - Type check all “executable” parts of the program at level  $\perp$  and “eraseable” parts *at least* at level  $C$
  - Noninterference tells us that erasure is *safe*

# Irrelevance for Dependent Types

Use lattice  $\perp < C < T$

- Dependency analysis for **run-time** irrelevance
  - What parts of the program can we safely *erase before execution*?
  - Type check all “executable” parts of the program at level  $\perp$  and “eraseable” parts *at least* at level  $C$
  - Noninterference tells us that erasure is *safe*
- Dependency analysis for **compile-time** irrelevance
  - What parts of the program can we safely *ignore when checking equivalence*?

# Irrelevance for Dependent Types

Use lattice  $\perp < C < \top$

- Dependency analysis for **run-time** irrelevance
  - What parts of the program can we safely *erase before execution*?
  - Type check all “executable” parts of the program at level  $\perp$  and “eraseable” parts *at least* at level  $C$
  - Noninterference tells us that erasure is *safe*
- Dependency analysis for **compile-time** irrelevance
  - What parts of the program can we safely *ignore when checking equivalence*?
  - Type check all “comparable” parts of the program at most at level  $C$  and all “ignorable” parts at level  $\top$

# Irrelevance for Dependent Types

Use lattice  $\perp < C < \top$

- Dependency analysis for **run-time** irrelevance
  - What parts of the program can we safely *erase before execution*?
  - Type check all “executable” parts of the program at level  $\perp$  and “eraseable” parts *at least* at level  $C$
  - Noninterference tells us that erasure is *safe*
- Dependency analysis for **compile-time** irrelevance
  - What parts of the program can we safely *ignore when checking equivalence*?
  - Type check all “comparable” parts of the program at most at level  $C$  and all “ignorable” parts at level  $\top$
  - Index definitional equality by observation level: equality at  $C$  cannot observe parts of the term marked at  $\top$

# Irrelevance for Dependent Types

Use lattice  $\perp < C < \top$

- Dependency analysis for **run-time** irrelevance
  - What parts of the program can we safely *erase before execution*?
  - Type check all “executable” parts of the program at level  $\perp$  and “eraseable” parts *at least* at level  $C$
  - Noninterference tells us that erasure is *safe*
- Dependency analysis for **compile-time** irrelevance
  - What parts of the program can we safely *ignore when checking equivalence*?
  - Type check all “comparable” parts of the program at most at level  $C$  and all “ignorable” parts at level  $\top$
  - Index definitional equality by observation level: equality at  $C$  cannot observe parts of the term marked at  $\top$
  - Noninterference tells us that indexed equality is *consistent*

# Example

Polymorphic identity function

$$\begin{aligned} \text{id} &:^\perp \prod x:^\top \text{Type}. x^\perp \rightarrow x \\ \text{id} &= \lambda^\top x. \lambda y^\perp. y \end{aligned}$$

Type parameter  $x$  is both erasable and ignorable.

Term parameter  $y$  is neither.



# Example

Polymorphic identity function

$$\begin{aligned} \text{id} &: {}^\perp \Pi x: {}^\top \text{Type}. x^\perp \rightarrow x \\ \text{id} &= \lambda^\top x. \lambda y^\perp. y \end{aligned}$$

Type parameter  $x$  is both erasable and ignorable.

Term parameter  $y$  is neither.

To decrease clutter in examples, elide  $\perp$  labels

$$\begin{aligned} \text{id} &: \Pi x: {}^\top \text{Type}. x \rightarrow x \\ \text{id} &= \lambda^\top x. \lambda y. y \end{aligned}$$

# Example

Polymorphic identity function

$$\text{id} : \prod x : \top \text{Type}. x \rightarrow x$$
$$\text{id} = \lambda^{\top} x. \lambda y. y$$

# Example

## Polymorphic identity function

$$\text{id} : \prod x : \top \text{Type}. x \rightarrow x$$
$$\text{id} = \lambda^\top x. \lambda y. y$$

- $\lambda$ -bound  $y$  (at level  $\perp$ ) can be used in the body of the function.
- $\lambda$ -bound  $x$  (at level  $\top$ ) cannot be used.

# Example

## Polymorphic identity function

$\text{id} : \Pi x : {}^\top\text{Type}. x \rightarrow x$

$\text{id} = \lambda^\top x. \lambda y. y$

- $\lambda$ -bound  $y$  (at level  $\perp$ ) can be used in the body of the function.
- $\lambda$ -bound  $x$  (at level  $\top$ ) cannot be used.
- Label  $\top$  on  $\Pi$ -bound  $x$  describes level of  $\lambda$ -bound  $x$ .
- $\Pi$ -bound  $x$  can be used in the body of the  $\Pi$ -type.

# Example

## Polymorphic identity function

$\text{id} : \Pi x : \top \text{Type}. x \rightarrow x$

$\text{id} = \lambda^{\top} x. \lambda y. y$

- $\lambda$ -bound  $y$  (at level  $\perp$ ) can be used in the body of the function.
- $\lambda$ -bound  $x$  (at level  $\top$ ) cannot be used.
- Label  $\top$  on  $\Pi$ -bound  $x$  describes level of  $\lambda$ -bound  $x$ .
- $\Pi$ -bound  $x$  can be used in the body of the  $\Pi$ -type.
- When evaluating  $\text{id } A^{\top} \text{ true}$  can erase argument  $A$
- During type checking, if comparing  $\text{id } A^{\top} \text{ true}$  and  $\text{id } B^{\top} \text{ true}$  for equality, can ignore  $A$  and  $B$

## Example: vectors (Haskell GADT-style)

```
Vec  : Nat -> Type -> Type
Nil  :  $\Pi n:\mathbb{T}Nat. \Pi a:\mathbb{T}Type. (n \sim Zero) \Rightarrow Vec\ n\ a$ 
Cons :  $\Pi n:\mathbb{T}Nat. \Pi a:\mathbb{T}Type. \Pi m:\mathbb{T}Nat. (n \sim Succ\ m) \Rightarrow$   
       $a \rightarrow Vec\ m\ a \rightarrow Vec\ n\ a$ 
```

- Applications of `Nil` and `Cons` can erase and ignore length and type parameters. (Will elide from examples.)
- Applications of `Vec` cannot. (Shouldn't equate vectors with different lengths/element types.)
- In type of `Nil` and `Cons`, `n` and `a` can be used freely.

## Example: vectors (Haskell GADT-style)

```
Vec  : Nat -> Type -> Type
Nil  :  $\Pi n:\top \text{Nat}. \Pi a:\top \text{Type}. (n \sim \text{Zero}) \Rightarrow \text{Vec } n \ a$ 
Cons :  $\Pi n:\top \text{Nat}. \Pi a:\top \text{Type}. \Pi m:\top \text{Nat}. (n \sim \text{Succ } m) \Rightarrow$   
       $a \rightarrow \text{Vec } m \ a \rightarrow \text{Vec } n \ a$ 
```

- Applications of Nil and Cons can erase and ignore length and type parameters. (Will elide from examples.)
- Applications of Vec cannot. (Shouldn't equate vectors with different lengths/element types.)
- In type of Nil and Cons, n and a can be used freely.

```
vmap :  $\Pi n:\top \text{Nat}. \Pi a \ b:\top \text{Type}. (a \rightarrow b) \rightarrow \text{Vec } n \ a \rightarrow \text{Vec } n \ b$ 
vmap =  $\lambda^\top n \ a \ b. \lambda f \ xs.$   
      case xs of  
        Nil -> Nil  
        Cons  $m^\top \ x \ xs \rightarrow \text{Cons } m^\top \ (f \ x) \ (\text{vmap } m^\top \ a^\top \ b^\top \ f \ xs)$ 
```

# Filter example

Suppose we have

```
a :  $\top$  Type      -- type of vector elements  
f : a  $\rightarrow$  Bool   -- predicate to filter with
```

Consider vector filter

```
filter :  $\prod n : \top \text{Nat}. \text{Vec } n \ a \rightarrow \sum m : \top \text{Nat}. \text{Vec } m \ a$ 
```



# Filter example

Suppose we have

```
a:  $\top$  Type      -- type of vector elements  
f: a -> Bool   -- predicate to filter with
```

Consider vector filter

```
filter :  $\prod n: \top \text{Nat}. \text{Vec } n \text{ a} \rightarrow \sum m: \top \text{Nat}. \text{Vec } m \text{ a}$   
filter =  $\lambda^\top n. \lambda \text{ vec}.$   
      case vec of  
        Nil -> (Zero $^\top$ , Nil)
```

# Filter example

Suppose we have

```
a:  $\top$  Type      -- type of vector elements  
f: a -> Bool   -- predicate to filter with
```

Consider vector filter

```
filter :  $\prod n: \top \text{Nat}. \text{Vec } n \text{ a} \rightarrow \sum m: \top \text{Nat}. \text{Vec } m \text{ a}$   
filter =  $\lambda \top n. \lambda \text{vec}.$   
  case vec of  
    Nil -> (Zero $^\top$ , Nil)  
    Cons n1 $^\top$  x xs  
      | f x      ->  
        let (m1 $^\top$ , v1) = filter n1 $^\top$  xs in
```

# Filter example

Suppose we have

```
a:  $\top$  Type      -- type of vector elements  
f: a -> Bool   -- predicate to filter with
```

Consider vector filter

```
filter :  $\prod n: \top \text{Nat}. \text{Vec } n \text{ a} \rightarrow \sum m: \top \text{Nat}. \text{Vec } m \text{ a}$   
filter =  $\lambda^\top n. \lambda \text{ vec}.$   
  case vec of  
    Nil -> (Zero $^\top$ , Nil)  
    Cons n1 $^\top$  x xs  
      | f x      ->  
        let (m1 $^\top$ , v1) = filter n1 $^\top$  xs in  
        ((Succ m1) $^\top$ , Cons m1 $^\top$  x v1)
```

# Filter example

Suppose we have

```
a:  $\top$  Type      -- type of vector elements  
f: a -> Bool   -- predicate to filter with
```

Consider vector filter

```
filter :  $\prod n: \top \text{Nat}. \text{Vec } n \text{ a} \rightarrow \sum m: \top \text{Nat}. \text{Vec } m \text{ a}$   
filter =  $\lambda^\top n. \lambda \text{ vec}.$   
  case vec of  
    Nil -> (Zero $^\top$ , Nil)  
    Cons n1 $^\top$  x xs  
      | f x      ->  
        let (m1 $^\top$ , v1) = filter n1 $^\top$  xs in  
        ((Succ m1) $^\top$ , Cons m1 $^\top$  x v1)  
      | otherwise -> filter n1 $^\top$  xs
```

# Filter example

Suppose we have

```
a:  $\top$  Type      -- type of vector elements  
f: a -> Bool    -- predicate to filter with
```

Consider vector filter

```
filter :  $\prod n: \top \text{Nat}. \text{Vec } n \text{ a} \rightarrow \sum m: \top \text{Nat}. \text{Vec } m \text{ a}$   
filter =  $\lambda^\top n. \lambda \text{ vec}.$   
  case vec of  
    Nil -> (Zero $^\top$ , Nil)  
    Cons n1 $^\top$  x xs  
      | f x      ->  
        let (m1 $^\top$ , v1) = filter n1 $^\top$  xs in  
        ((Succ m1) $^\top$ , Cons m1 $^\top$  x v1)  
      | otherwise -> filter n1 $^\top$  xs
```

This version is overly strict. Must filter entire list before returning anything.

# Filter example

Suppose we have

$$\text{fst} : \Sigma x:\ell A. B \rightarrow A$$
$$\text{snd} : \Pi p:(\Sigma x:\ell A. B). B \{ \text{fst } p / x \}$$

# Filter example

Suppose we have

$$\text{fst} : \sum x : \ell A. B \rightarrow A$$
$$\text{snd} : \prod p : (\sum x : \ell A. B). B \{ \text{fst } p / x \}$$
$$\text{filter} : \prod n : \top \text{Nat}. \text{Vec } n \ a \rightarrow \sum m : \text{Nat}. \text{Vec } m \ a$$
$$\text{filter} = \lambda^\top n. \lambda \text{vec}.$$
$$\text{case vec of}$$
$$\text{Nil} \rightarrow (\text{Zero}, \text{Nil})$$
$$\text{Cons } n1^\top \ x \ xs$$
$$\mid f \ x \quad \rightarrow$$
$$((\text{Succ } (\text{fst } ys)), \text{Cons } (\text{fst } ys)^\top \ x \ (\text{snd } ys))$$
$$\text{where}$$
$$ys : \sum m : \text{Nat}. \text{Vec } m \ a$$
$$ys = \text{filter } n1^\top \ xs$$
$$\mid \text{otherwise} \rightarrow \text{filter } n1^\top \ xs$$

# Filter example

Suppose we have

$$\text{fst} : \Sigma x : \ell A. B \rightarrow A$$
$$\text{snd} : \prod p : (\Sigma x : \ell A. B). B \{ \text{fst } p / x \}$$
$$\text{filter} : \prod n : \top \text{Nat}. \text{Vec } n \ a \rightarrow \Sigma m : \text{Nat}. \text{Vec } m \ a$$
$$\text{filter} = \lambda^\top n. \lambda \text{vec}.$$
$$\text{case vec of}$$
$$\text{Nil} \rightarrow (\text{Zero}, \text{Nil})$$
$$\text{Cons } n1^\top \ x \ xs$$
$$\mid f \ x \quad \rightarrow$$
$$((\text{Succ } (\text{fst } ys)), \text{Cons } (\text{fst } ys)^\top \ x \ (\text{snd } ys))$$
$$\text{where}$$
$$ys : \Sigma m : \text{Nat}. \text{Vec } m \ a$$
$$ys = \text{filter } n1^\top \ xs$$
$$\mid \text{otherwise} \rightarrow \text{filter } n1^\top \ xs$$

Can we mark  $m$  in the  $\Sigma$ -type as  $\top$  (ignorable)?



# Filter example

Suppose we have

$$\text{fst} : \Sigma x : \ell A. B \rightarrow A$$
$$\text{snd} : \prod p : (\Sigma x : \ell A. B). B \{ \text{fst } p / x \}$$
$$\text{filter} : \prod n : \top \text{Nat}. \text{Vec } n \ a \rightarrow \Sigma m : \text{Nat}. \text{Vec } m \ a$$
$$\text{filter} = \lambda^\top n. \lambda \text{vec}.$$
$$\text{case vec of}$$
$$\text{Nil} \rightarrow (\text{Zero}, \text{Nil})$$
$$\text{Cons } n1^\top \ x \ xs$$
$$\mid f \ x \quad \rightarrow$$
$$((\text{Succ } (\text{fst } ys)), \text{Cons } (\text{fst } ys)^\top \ x \ (\text{snd } ys))$$
$$\text{where}$$
$$ys : \Sigma m : \text{Nat}. \text{Vec } m \ a$$
$$ys = \text{filter } n1^\top \ xs$$
$$\mid \text{otherwise} \rightarrow \text{filter } n1^\top \ xs$$

Can we mark  $m$  in the  $\Sigma$ -type as  $\top$  (ignorable)?

**No!**  $\text{fst } ys$  cannot be ignored in the type of  $\text{snd } ys$ .

# Filter example

Use of  $C$  to mark erasable but not ignorable data.

```
filter :  $\prod n : \mathbb{N}^{\top} \text{Nat}. \text{Vec } n \text{ a} \rightarrow \Sigma m : C \text{Nat}. \text{Vec } m \text{ a}$   
filter =  $\lambda^{\top} n. \lambda \text{vec}.$   
  case vec of  
    Nil  $\rightarrow (\text{Zero}^C, \text{Nil})$   
    Cons  $n1^{\top} \text{ x xs}$   
      | f x  $\rightarrow$   
        ((Succ (fst ys)) $^C$ , Cons (fst ys) $^{\top} \text{ x (snd ys)}$ )  
        where  
          ys = filter  $n1^{\top} \text{ xs}$   
      | otherwise  $\rightarrow$  filter  $n1^{\top} \text{ xs}$ 
```

Three levels provides us with the precision that we need to write this code.

# Type system in depth

T-ABS

$$\frac{\begin{array}{c} \Gamma, x : {}^{\ell_0 \vee \ell} A \vdash b : {}^{\ell} B \\ C \wedge \Gamma \vdash (\Pi x : {}^{\ell_0} A. B) : {}^C s \end{array}}{\Gamma \vdash \lambda x : {}^{\ell_0} A. b : {}^{\ell} \Pi x : {}^{\ell_0} A. B}$$

T-APPC

$$\frac{\Gamma \vdash b : {}^{\ell} \Pi x : {}^{\ell_0} A. B \quad \Gamma \Vdash a : {}^{\ell_0 \vee \ell} A}{\Gamma \vdash b \ a^{\ell_0} : {}^{\ell} B\{a/x\}}$$

T-PI

$$\frac{\begin{array}{c} \Gamma \vdash A : {}^{\ell} s_1 \\ \Gamma, x : {}^{\ell} A \vdash B : {}^{\ell} s_2 \\ \mathcal{R}(s_1, s_2, s_3) \end{array}}{\Gamma \vdash \Pi x : {}^{\ell_0} A. B : {}^{\ell} s_3}$$

- Use  $\Gamma \vdash a : {}^{\top} s$  to check  $\Pi$ -type in rule T-ABS.

# Type system in depth

T-ABS

$$\frac{\begin{array}{c} \Gamma, x : {}^{\ell_0 \vee \ell} A \vdash b : {}^\ell B \\ C \wedge \Gamma \vdash (\Pi x : {}^{\ell_0} A. B) : {}^C s \end{array}}{\Gamma \vdash \lambda x : {}^{\ell_0} A. b : {}^\ell \Pi x : {}^{\ell_0} A. B}$$

T-APPC

$$\frac{\Gamma \vdash b : {}^\ell \Pi x : {}^{\ell_0} A. B \quad \Gamma \Vdash a : {}^{\ell_0 \vee \ell} A}{\Gamma \vdash b \ a^{\ell_0} : {}^\ell B\{a/x\}}$$

T-PI

$$\frac{\begin{array}{c} \Gamma \vdash A : {}^\ell s_1 \\ \Gamma, x : {}^\ell A \vdash B : {}^\ell s_2 \\ \mathcal{R}(s_1, s_2, s_3) \end{array}}{\Gamma \vdash \Pi x : {}^{\ell_0} A. B : {}^\ell s_3}$$

- Use  $\Gamma \vdash a : {}^\top s$  to check  $\Pi$ -type in rule T-ABS.
- $\Pi x : {}^\ell A. B$  acts a little like  $\Pi x : (T^\ell A). B$ , so rule T-ABS looks like rule SDC-BIND and rule T-APPC looks like rule SDC-RETURN.

# Type system in depth

T-ABS

$$\frac{\begin{array}{c} \Gamma, x :^{\ell_0 \vee \ell} A \vdash b :^\ell B \\ C \wedge \Gamma \vdash (\Pi x :^{\ell_0} A. B) :^C s \end{array}}{\Gamma \vdash \lambda x :^{\ell_0} A. b :^\ell \Pi x :^{\ell_0} A. B}$$

T-APPC

$$\frac{\Gamma \vdash b :^\ell \Pi x :^{\ell_0} A. B \quad \Gamma \Vdash a :^{\ell_0 \vee \ell} A}{\Gamma \vdash b \ a^{\ell_0} :^\ell B\{a/x\}}$$

T-PI

$$\frac{\begin{array}{c} \Gamma \vdash A :^\ell s_1 \\ \Gamma, x :^\ell A \vdash B :^\ell s_2 \\ \mathcal{R}(s_1, s_2, s_3) \end{array}}{\Gamma \vdash \Pi x :^{\ell_0} A. B :^\ell s_3}$$

- Use  $\Gamma \vdash a :^\top s$  to check  $\Pi$ -type in rule T-ABS.
- $\Pi x :^\ell A. B$  acts a little like  $\Pi x : (T^\ell A). B$ , so rule T-ABS looks like rule SDC-BIND and rule T-APPC looks like rule SDC-RETURN.
- Important difference:  $x$  labelled with  $\ell$  instead of  $\ell_0 \vee \ell$  in rule T-PI.

# Type system in depth

T-ABS

$$\frac{\begin{array}{c} \Gamma, x :^{\ell_0 \vee \ell} A \vdash b :^\ell B \\ C \wedge \Gamma \vdash (\Pi x :^{\ell_0} A. B) :^C s \end{array}}{\Gamma \vdash \lambda x :^{\ell_0} A. b :^\ell \Pi x :^{\ell_0} A. B}$$

T-APPC

$$\frac{\Gamma \vdash b :^\ell \Pi x :^{\ell_0} A. B \quad \Gamma \Vdash a :^{\ell_0 \vee \ell} A}{\Gamma \vdash b \ a^{\ell_0} :^\ell B\{a/x\}}$$

T-PI

$$\frac{\begin{array}{c} \Gamma \vdash A :^\ell s_1 \\ \Gamma, x :^\ell A \vdash B :^\ell s_2 \\ \mathcal{R}(s_1, s_2, s_3) \end{array}}{\Gamma \vdash \Pi x :^{\ell_0} A. B :^\ell s_3}$$

- Use  $\Gamma \vdash a :^\top s$  to check  $\Pi$ -type in rule T-ABS.
- $\Pi x :^\ell A. B$  acts a little like  $\Pi x : (T^\ell A). B$ , so rule T-ABS looks like rule SDC-BIND and rule T-APPC looks like rule SDC-RETURN.
- Important difference:  $x$  labelled with  $\ell$  instead of  $\ell_0 \vee \ell$  in rule T-PI.
- To know result type of rule T-APPC is well-formed, have  $C \wedge \Gamma, x :^C A \vdash B :^C s$ , so label of  $a$  must be  $\leq C$ , motivating use of “resurrection”

DDC is only type system with multiple, independent levels of irrelevance. This distinction is essential for strong  $\Sigma$ -types with erasable first components.

- *Both run-time and compile-time irrelevance, but no distinction between them.* ICC (Miquel 2001, Barras and Bernardo 2009), Mishra-Linger Sheard (2008), Dependent Haskell (2017). Implicit version omits irrelevant data. Explicit version relies on erasure.

DDC is only type system with multiple, independent levels of irrelevance. This distinction is essential for strong  $\Sigma$ -types with erasable first components.

- *Both run-time and compile-time irrelevance, but no distinction between them.* ICC (Miquel 2001, Barras and Bernardo 2009), Mishra-Linger Sheard (2008), Dependent Haskell (2017). Implicit version omits irrelevant data. Explicit version relies on erasure.
- *Run-time irrelevance only.* Brady (2004, 2013). Quantitative type theory (McBride 2016, Atkey 2018). Generalizes to arbitrary semiring, but does not track irrelevance in types. Tejiščák (2020) notes that erasure should be different from ignorability, but only supports erasure.



DDC is only type system with multiple, independent levels of irrelevance. This distinction is essential for strong  $\Sigma$ -types with erasable first components.

- *Both run-time and compile-time irrelevance, but no distinction between them.* ICC (Miquel 2001, Barras and Bernardo 2009), Mishra-Linger Sheard (2008), Dependent Haskell (2017). Implicit version omits irrelevant data. Explicit version relies on erasure.
- *Run-time irrelevance only.* Brady (2004, 2013). Quantitative type theory (McBride 2016, Atkey 2018). Generalizes to arbitrary semiring, but does not track irrelevance in types. Tejiščák (2020) notes that erasure should be different from ignorability, but only supports erasure.
- *Compile-time irrelevance only.* Pfenning (2001), Abel and Scherer (2012). Type-sensitive definitional equivalence, so fewer arguments can be ignored in types. Usage of variable in  $\Pi$  must match use in  $\lambda$ .

# Conclusion

- We have syntactic proofs of noninterference and type soundness for DDC, formalized using Coq  
<http://github.com/sweirich/graded-haskell/>
- These proofs are for an arbitrary pure type system and do not require the type system to be strongly normalizing. Future work: Prove consistency and decidable type checking for some instance of DDC.
- In DDC, indexed definitional equality is untyped. Future work: use a typed equality and type-directed equivalence.
- Type system is general enough to support lattice of run-time security levels below  $C$ . Future work: propositional form of indexed equivalence for reasoning about security-typed programs.

# Backup slides

# Typing rules for DDC

$$\boxed{\Gamma \vdash a :^\ell A}$$

(DDC typing rules)

$$\begin{array}{c} \text{T-VAR} \\ \ell_0 \leq \ell \\ x :^{\ell_0} A \in \Gamma \\ \ell \leq C \\ \hline \Gamma \vdash x :^\ell A \end{array}$$

$$\begin{array}{c} \text{T-PI} \\ \Gamma \vdash A :^\ell s_1 \\ \Gamma, x :^\ell A \vdash B :^\ell s_2 \\ \mathcal{R}(s_1, s_2, s_3) \\ \hline \Gamma \vdash \Pi x :^{\ell_0} A. B :^\ell s_3 \end{array}$$

$$\begin{array}{c} \text{T-ABSC} \\ \Gamma, x :^{\ell_0 \vee \ell} A \vdash b :^\ell B \\ \Gamma \Vdash (\Pi x :^{\ell_0} A. B) :^\top s \\ \hline \Gamma \vdash \lambda x :^{\ell_0} A. b :^\ell \Pi x :^{\ell_0} A. B \end{array}$$

$$\begin{array}{c} \text{T-APPC} \\ \Gamma \vdash b :^\ell \Pi x :^{\ell_0} A. B \\ \Gamma \Vdash a :^{\ell_0 \vee \ell} A \\ \hline \Gamma \vdash b a^{\ell_0} :^\ell B\{a/x\} \end{array}$$

$$\begin{array}{c} \text{T-CONVC} \\ \Gamma \vdash a :^\ell A \\ |C \wedge \Gamma| \vdash A \equiv_C B \\ \Gamma \Vdash B :^\top s \\ \hline \Gamma \vdash a :^\ell B \end{array}$$

$$\begin{array}{c} \text{T-TYPE} \\ \ell \leq C \quad \mathcal{A}(s_1, s_2) \\ \hline \Gamma \vdash s_1 :^\ell s_2 \end{array}$$

# Typing rules for DDC (continued)

$$\boxed{\Gamma \Vdash a :^\ell A}$$

(*Truncate at  $\top$* )

CT-LEQ

$$\frac{\Gamma \vdash a :^\ell A \quad \ell \leq C}{\Gamma \Vdash a :^\ell A}$$

CT-TOP

$$\frac{C \wedge \Gamma \vdash a :^C A \quad C < \ell}{\Gamma \Vdash a :^\ell A}$$

# Typing rules for $\Sigma$ -types

T-SPAIR

$$\frac{C \wedge \Gamma \vdash \Sigma x:\ell_0 A.B :^C s \quad \Gamma \vdash a :^{\ell_0 \vee \ell} A \quad \Gamma \vdash b :^\ell B\{a/x\} \quad \ell_0 \leq C}{\Gamma \vdash (a^{\ell_0}, b) :^\ell \Sigma x:\ell_0 A.B}$$

T-LETPAIRC

$$\frac{\Gamma \vdash a :^\ell \Sigma x:\ell_0 A.B \quad \Gamma, x:\ell_0 \vee \ell A, y:\ell B \vdash c :^\ell C\{(x^{\ell_0}, y)/z\} \quad \Gamma, z:^\top (\Sigma x:\ell_0 A.B) \Vdash C :^\top s}{\Gamma \vdash \mathbf{let} (x^{\ell_0}, y) = a \mathbf{in} c :^\ell C\{a/z\}}$$

T-PROJ1

$$\frac{\Gamma \vdash a :^\ell \Sigma x:\ell_0 A.B \quad \ell_0 \leq \ell}{\Gamma \vdash \pi_1^{\ell_0} a :^\ell A}$$

T-PROJ2

$$\frac{\Gamma \vdash a :^\ell \Sigma x:\ell_0 A.B \quad \ell_0 \leq C}{\Gamma \vdash \pi_2^{\ell_0} a :^\ell B\{\pi_1^{\ell_0} a/x\}}$$