A Dependent Dependency Calculus

Pritam Choudhury
Harley Eades III
Stephanie Weirich

June 12, 2022

Dependency Analysis

- Consider

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

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

Dependency Analysis

- Consider

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

  where type system 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.

Dependency Analysis

- Consider

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

  where type system 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

$\boxed{\Gamma \vdash a :^{\ell} A}$ *(Simple types)*

$$\textsc{SDC-Var}\quad \frac{\ell_0 \leq \ell \qquad x :^{\ell_0} A \in \Gamma}{\Gamma \vdash x :^{\ell} A}$$

$$\textsc{SDC-Abs}\quad \frac{\Gamma, x :^{\ell} A \vdash b :^{\ell} B}{\Gamma \vdash \lambda x : A . b :^{\ell} A \to B}$$

$$\textsc{SDC-App}\quad \frac{\Gamma \vdash b :^{\ell} A \to B \qquad \Gamma \vdash a :^{\ell} A}{\Gamma \vdash b\ a :^{\ell} B}$$

Dependency and simple types

$\boxed{\Gamma \vdash a :^{\ell} A}$ *(Simple types)*

$$\frac{\begin{array}{c}\ell_0 \leq \ell \\ x :^{\ell_0} A \in \Gamma\end{array}}{\Gamma \vdash x :^{\ell} A}\ \textsc{SDC-Var} \qquad \frac{\Gamma, x :^{\ell} A \vdash b :^{\ell} B}{\Gamma \vdash \lambda x : A.b :^{\ell} A \to B}\ \textsc{SDC-Abs} \qquad \frac{\begin{array}{c}\Gamma \vdash b :^{\ell} A \to B \\ \Gamma \vdash a :^{\ell} A\end{array}}{\Gamma \vdash b\ a :^{\ell} B}\ \textsc{SDC-App}$$

Internalize judgment with graded modal type
$T^{\ell_0}\ A$ describes terms of type A checked at least at level ℓ_0

Dependency and simple types

$\boxed{\Gamma \vdash a :^{\ell} A}$ *(Simple types)*

$$\frac{\ell_0 \leq \ell \qquad x :^{\ell_0} A \in \Gamma}{\Gamma \vdash x :^{\ell} A}\ \textsc{SDC-Var}$$

$$\frac{\Gamma, x :^{\ell} A \vdash b :^{\ell} B}{\Gamma \vdash \lambda x : A . b :^{\ell} A \to B}\ \textsc{SDC-Abs}$$

$$\frac{\Gamma \vdash b :^{\ell} A \to B \qquad \Gamma \vdash a :^{\ell} A}{\Gamma \vdash b\ a :^{\ell} B}\ \textsc{SDC-App}$$

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

$$\frac{\Gamma \vdash a :^{\ell} T^{\ell_0} A \qquad \Gamma, x :^{\ell \vee \ell_0} A \vdash b :^{\ell} B}{\Gamma \vdash \mathbf{bind}^{\ell_0}\, x = a\, \mathbf{in}\, b :^{\ell} B}\ \textsc{SDC-Bind}$$

Dependency and simple types

$\boxed{\Gamma \vdash a :^{\ell} A}$ *(Simple types)*

$$\textsc{SDC-Var}\quad \frac{\ell_0 \leq \ell \qquad x :^{\ell_0} A \in \Gamma}{\Gamma \vdash x :^{\ell} A}$$

$$\textsc{SDC-Abs}\quad \frac{\Gamma, x :^{\ell} A \vdash b :^{\ell} B}{\Gamma \vdash \lambda x : A.b :^{\ell} A \to B}$$

$$\textsc{SDC-App}\quad \frac{\Gamma \vdash b :^{\ell} A \to B \qquad \Gamma \vdash a :^{\ell} A}{\Gamma \vdash b\ a :^{\ell} B}$$

$$\textsc{SDC-Return}\quad \frac{\Gamma \vdash a :^{\ell \vee \ell_0} A}{\Gamma \vdash \eta^{\ell_0}\ a :^{\ell} T^{\ell_0}\ A}$$

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

No need for DCC's projection judgement due to label ℓ.

Dependency and simple types

$\boxed{\Gamma \vdash a :^{\ell} A}$ *(Simple types)*

$$\textsc{SDC-Var}\ \frac{\ell_0 \leq \ell \qquad x :^{\ell_0} A \in \Gamma}{\Gamma \vdash x :^{\ell} A}$$

$$\textsc{SDC-Abs}\ \frac{\Gamma, x :^{\ell} A \vdash b :^{\ell} B}{\Gamma \vdash \lambda x : A . b :^{\ell} A \to B}$$

$$\textsc{SDC-App}\ \frac{\Gamma \vdash b :^{\ell} A \to B \qquad \Gamma \vdash a :^{\ell} A}{\Gamma \vdash b\ a :^{\ell} B}$$

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

$$\textsc{SDC-Bind}\ \frac{\Gamma \vdash a :^{\ell} T^{\ell_0}\ A \qquad \Gamma, x :^{\ell \vee \ell_0} A \vdash b :^{\ell} B}{\Gamma \vdash \mathbf{bind}^{\ell_0}\, x = a\, \mathbf{in}\, b :^{\ell} B}$$

No need for DCC's projection judgement due to label ℓ.
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)*

$$\textsc{SDC-Var}\quad \frac{\ell_0 \leq \ell \qquad x :^{\ell_0} A \in \Gamma}{\Gamma \vdash x :^{\ell} A}$$

$$\textsc{SDC-Abs}\quad \frac{\Gamma, x :^{\ell} A \vdash b :^{\ell} B}{\Gamma \vdash \lambda x : A . b :^{\ell} A \to B}$$

$$\textsc{SDC-App}\quad \frac{\Gamma \vdash b :^{\ell} A \to B \qquad \Gamma \vdash a :^{\ell} A}{\Gamma \vdash b\ a :^{\ell} B}$$

$$\textsc{SDC-Return}\quad \frac{\Gamma \vdash a :^{\ell \vee \ell_0} A}{\Gamma \vdash \eta^{\ell_0}\ a :^{\ell} T^{\ell_0}\ A}$$

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

$$\textsc{Sealing-Unseal}\quad \frac{\Gamma \vdash a :^{\ell} T^{\ell_0}\ A \qquad \ell_0 \leq \ell}{\Gamma \vdash \mathbf{unseal}^{\ell_0}\, a :^{\ell} A}$$

Indexed indistinguishability

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

- a and b differ only in places marked by η^{ℓ_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 η^{ℓ_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 η^{ℓ_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 η^{ℓ_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* $a_1 \leadsto a_2$ *then there exists some* a_2' *such that* $a_1' \leadsto a_2'$ *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* $\varnothing \vdash a_1, a_2 :^H\ A$, *if* $\vdash b\{a_1/x\} \rightsquigarrow^* v_1$ *and* $\vdash b\{a_2/x\} \rightsquigarrow^* v_2$ *then* $v_1 = v_2$.

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

This follows because we can show that
$\varnothing \vdash\ \lambda y.\mathbf{bind}\, x = y\, \mathbf{in}\, b\ :^L\ T^H\ A \to \mathbf{Int}$ and
$\varnothing \vdash (\lambda y.\mathbf{bind}\, x = y\, \mathbf{in}\, b)\ (\eta^H\ a_1) \sim_L (\lambda y.\mathbf{bind}\, x = y\, \mathbf{in}\, b)\ (\eta^H\ a_2)$.
Type soundness says that if both terms terminate, then both must be integer values. Theorem above states that they must be the same integer.

Label-indexed definitional equality

Define *label-indexed definitional equality*, $\Phi \vdash a \equiv_{\ell} b$ as the closure of indexed indistinguishability by β-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 ℓ
- Π and Σ 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 ℓ
- Π and Σ types annotated with levels ($\Pi x :^{\ell} A.B$ and $\Sigma x{:}^{\ell} A.B$)
- Σ 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 ℓ
- Π and Σ types annotated with levels ($\Pi x :^{\ell} A.B$ and $\Sigma x{:}^{\ell}A.B$)
- Σ types encode labelled graded type: $T^{\ell}\ A \triangleq \Sigma x{:}^{\ell}A.\mathbf{unit}$
- Lattice must have a distinguished element C, with $\bot \leq C \leq \top$

Dependent Dependency Calculus (DDC)

- DDC is a pure type system extended with an arbitrary lattice of dependency levels ℓ
- Π and Σ types annotated with levels ($\Pi x :^{\ell} A.B$ and $\Sigma x{:}^{\ell}A.B$)
- Σ types encode labelled graded type: $T^{\ell}\ A \triangleq \Sigma x{:}^{\ell}A.\mathbf{unit}$
- Lattice must have a distinguished element C, with $\bot \leq C \leq \top$
- Definitional equality is indexed by C (i.e. we use $\Phi \vdash a \equiv_C b$)

Dependent Dependency Calculus (DDC)

- DDC is a pure type system extended with an arbitrary lattice of dependency levels ℓ
- Π and Σ types annotated with levels ($\Pi x :^{\ell} A.B$ and $\Sigma x{:}^{\ell} A.B$)
- Σ types encode labelled graded type: $T^{\ell}\ A \triangleq \Sigma x{:}^{\ell} A.\mathbf{unit}$
- Lattice must have a distinguished element C, with $\bot \leq C \leq \top$
- Definitional equality is indexed by C (i.e. we use $\Phi \vdash a \equiv_C b$)
- Dependency levels track phase
  - Executable: $\Gamma \vdash a :^{\bot} A$
  - Comparable: $\Gamma \vdash a :^{C} A$
  - Irrelevant: $\Gamma \vdash a :^{\top} A$
    (When $C \neq \top$, defined as $C \wedge \Gamma \vdash a :^{C} A$)

Dependent Dependency Calculus (DDC)

- DDC is a pure type system extended with an arbitrary lattice of dependency levels ℓ
- Π and Σ types annotated with levels ($\Pi x :^{\ell} A.B$ and $\Sigma x{:}^{\ell} A.B$)
- Σ types encode labelled graded type: $T^{\ell}\ A \triangleq \Sigma x{:}^{\ell} A.\mathbf{unit}$
- Lattice must have a distinguished element C, with $\bot \leq C \leq \top$
- Definitional equality is indexed by C (i.e. we use $\Phi \vdash a \equiv_C b$)
- Dependency levels track phase
  - Executable: $\Gamma \vdash a :^{\bot} A$
  - Comparable: $\Gamma \vdash a :^{C} A$
  - Irrelevant: $\Gamma \vdash a :^{\top} A$
    (When $C \neq \top$, defined as $C \wedge \Gamma \vdash a :^{C} A$)
- Results about DDC (noninterference, type soundness) proven in Coq and validated by artifact evaluation
  `https://github.com/sweirich/graded-haskell`

Irrelevance for Dependent Types

Use lattice $\bot < C < \top$

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

Irrelevance for Dependent Types

Use lattice $\bot < C < \top$

- 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 $\bot$ and "eraseable" parts *at least* at level C

Irrelevance for Dependent Types

Use lattice $\bot < C < \top$

- 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 $\bot$ and "eraseable" parts *at least* at level C
  - Noninterference tells us that erasure is *safe*

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*?

Irrelevance for Dependent Types

Use lattice $\bot < C < \top$

- 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 $\bot$ 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 $\bot < C < \top$

- 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 $\bot$ 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 $\bot < C < \top$

- 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 $\bot$ 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

$\texttt{id} :^{\perp} \Pi\ \texttt{x}:^{\top}\texttt{Type.}\ \texttt{x}^{\perp}\ \texttt{->}\ \texttt{x}$

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

Type parameter `x` is both eraseable and ignorable.
Term parameter `y` is neither.

Example

Polymorphic identity function

```
id :⊥ Π x:⊤Type. x⊥ -> x
id = λ⊤x. λy⊥. y
```

$\texttt{id} :^{\perp} \Pi\ \texttt{x}:^{\top}\texttt{Type.}\ \texttt{x}^{\perp}\ \texttt{->}\ \texttt{x}$

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

Type parameter `x` is both eraseable and ignorable.
Term parameter `y` is neither.

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

$\texttt{id} : \Pi\ \texttt{x}:^{\top}\texttt{Type.}\ \texttt{x}\ \texttt{->}\ \texttt{x}$

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

Example

Polymorphic identity function

```
id : Π x:^⊤Type. x -> x
id = λ^⊤x. λy. y
```

Example

Polymorphic identity function

```
id : Π x:⊤Type. x -> x
id = λ⊤x. λy. y
```

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

Example

Polymorphic identity function

```
id : Π x:⊤Type. x -> x
id = λ⊤x. λy. y
```

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

- Label $\top$ on Π-bound x describes level of λ-bound x.
- Π-bound x can be used in the body of the Π-type.

Example

Polymorphic identity function

```
id : Π x:⊤Type. x -> x
id = λ⊤x. λy. y
```

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

Example: vectors (Haskell GADT-style)

```
Vec  : Nat -> Type -> Type
Nil  : Π n:⊤Nat. Π a:⊤Type. (n ∼ Zero) => Vec n a
Cons : Π n:⊤Nat. Π a:⊤Type. Π m:⊤Nat.  (n ∼ Succ m) =>
       a -> Vec m a -> 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)

$$
\begin{array}{l}
\texttt{Vec\ \ : Nat -> Type -> Type} \\
\texttt{Nil\ \ : }\Pi\ \texttt{n:}^{\top}\texttt{Nat.}\ \Pi\ \texttt{a:}^{\top}\texttt{Type.}\ \texttt{(n} \sim \texttt{Zero) => Vec n a} \\
\texttt{Cons : }\Pi\ \texttt{n:}^{\top}\texttt{Nat.}\ \Pi\ \texttt{a:}^{\top}\texttt{Type.}\ \Pi\ \texttt{m:}^{\top}\texttt{Nat.}\ \ \texttt{(n} \sim \texttt{Succ m) =>} \\
\qquad\quad \texttt{a -> Vec m a -> Vec n a}
\end{array}
$$

- 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.

$$
\begin{array}{l}
\texttt{vmap : }\Pi\ \texttt{n:}^{\top}\texttt{Nat.}\Pi\ \texttt{a b:}^{\top}\texttt{Type.}\ \texttt{(a -> b) -> Vec n a -> Vec n b} \\
\texttt{vmap = }\lambda^{\top}\ \texttt{n a b.}\ \lambda\ \texttt{f xs.} \\
\qquad\qquad \texttt{case xs of} \\
\qquad\qquad\quad \texttt{Nil -> Nil} \\
\qquad\qquad\quad \texttt{Cons m}^{\top}\ \texttt{x xs -> Cons m}^{\top}\ \texttt{(f x) (vmap m}^{\top}\ \texttt{a}^{\top}\ \texttt{b}^{\top}\ \texttt{f xs)}
\end{array}
$$

Filter example

Suppose we have

```
a:⊤ Type      -- type of vector elements, eraseable
f: a -> Bool  -- predicate to filter with
```

Consider vector filter

```
filter : Πn:⊤Nat. Vec n a -> Σm:⊤Nat. Vec m a
```

Filter example

Suppose we have

```
a:⊤ Type       -- type of vector elements, eraseable
f: a -> Bool  -- predicate to filter with
```

Consider vector filter

```
filter : Πn:⊤Nat. Vec n a -> Σm:⊤Nat. Vec m a
filter = λ⊤ n. λ vec.
           case vec of
             Nil -> (Zero⊤, Nil)
```

Filter example

Suppose we have

```
a:⊤ Type         -- type of vector elements, eraseable
f: a -> Bool    -- predicate to filter with
```

Consider vector filter

```
filter : Πn:⊤Nat. Vec n a -> Σm:⊤Nat. Vec m a
filter = λ⊤ n. λ vec.
          case vec of
            Nil -> (Zero⊤, Nil)
            Cons n1⊤ x xs
              | f x        ->
                 let (m1⊤, v1) = filter n1⊤ xs in
```

Filter example

Suppose we have

```
a:⊤ Type        -- type of vector elements, eraseable
f: a -> Bool   -- predicate to filter with
```

Consider vector filter

```
filter : Πn:⊤Nat. Vec n a -> Σm:⊤Nat. Vec m a
filter = λ⊤ n. λ vec.
          case vec of
            Nil -> (Zero⊤, Nil)
            Cons n1⊤ x xs
              | f x        ->
                  let (m1⊤, v1) = filter n1⊤ xs in
                  ((Succ m1)⊤, Cons m1⊤ x v1)
```

Filter example

Suppose we have

```
a:⊤ Type       -- type of vector elements, eraseable
f: a -> Bool  -- predicate to filter with
```

Consider vector filter

```
filter : Πn:⊤Nat. Vec n a -> Σm:⊤Nat. Vec m a
filter = λ⊤ n. λ vec.
           case vec of
             Nil -> (Zero⊤, Nil)
             Cons n1⊤ x xs
               | f x         ->
                   let (m1⊤, v1) = filter n1⊤ xs in
                   ((Succ m1)⊤, Cons m1⊤ x v1)
               | otherwise -> filter n1⊤ xs
```

Filter example

Suppose we have

```
a:⊤ Type       -- type of vector elements, eraseable
f: a -> Bool  -- predicate to filter with
```

Consider vector filter

```
filter : Πn:⊤Nat. Vec n a -> Σm:⊤Nat. Vec m a
filter = λ⊤ n. λ vec.
           case vec of
             Nil -> (Zero⊤, Nil)
             Cons n1⊤ x xs
               | f x         ->
                  let (m1⊤, v1) = filter n1⊤ xs in
                  ((Succ m1)⊤, Cons m1⊤ x v1)
               | otherwise -> filter n1⊤ xs
```

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

Filter example

Suppose we have

```
fst : Σx:ℓA.B -> A
snd : Πp:(Σx:ℓA.B). B { fst p / x }
```

Filter example

Suppose we have

```
fst : Σx:ℓA.B -> A
snd : Πp:(Σx:ℓA.B). B { fst p / x }

filter : Πn:⊤Nat. Vec n a -> Σm:Nat. Vec m a
filter = λ⊤ n. λ vec.
          case vec of
            Nil -> (Zero, Nil)
            Cons n1⊤ x xs
              | f x         ->
                 ((Succ (fst ys)), Cons (fst ys)⊤ x (snd ys))
                     where
                       ys : Σ m : Nat. Vec m a
                       ys = filter n1⊤ xs
              | otherwise -> filter n1⊤ xs
```

Filter example

Suppose we have

```
fst : Σx:^ℓ A.B -> A
snd : Πp:(Σx:^ℓ A.B). B { fst p / x }

filter : Πn:^⊤ Nat. Vec n a -> Σm:Nat. Vec m a
filter = λ^⊤ n. λ vec.
          case vec of
            Nil -> (Zero, Nil)
            Cons n1^⊤ x xs
              | f x        ->
                 ((Succ (fst ys)), Cons (fst ys)^⊤ x (snd ys))
                     where
                       ys : Σ m : Nat. Vec m a
                       ys = filter n1^⊤ xs
              | otherwise -> filter n1^⊤ xs
```

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

Filter example

Suppose we have

```
fst : Σx:^ℓA.B -> A
snd : Πp:(Σx:^ℓA.B). B { fst p / x }

filter : Πn:^⊤Nat. Vec n a -> Σm:Nat. Vec m a
filter = λ^⊤ n. λ vec.
          case vec of
            Nil -> (Zero, Nil)
            Cons n1^⊤ x xs
              | f x        ->
                 ((Succ (fst ys)), Cons (fst ys)^⊤ x (snd ys))
                     where
                       ys : Σ m : Nat. Vec m a
                       ys = filter n1^⊤ xs
              | otherwise -> filter n1^⊤ xs
```

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

No! `fst ys` cannot be ignored in the type of `snd ys`.

Filter example

Use of C to mark eraseable but not ignorable data.

```
filter : Πn:⊤Nat. Vec n a -> Σm:CNat. Vec m a
filter = λ⊤ n. λ vec.
          case vec of
            Nil -> (ZeroC, Nil)
            Cons n1⊤ x xs
              | f x        ->
                 ((Succ (fst ys))C, Cons (fst ys)⊤ x (snd ys))
                     where
                       ys = filter n1⊤ xs
              | otherwise -> filter n1⊤ xs
```

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

Type system in depth

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

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

$$\frac{\Gamma \vdash b :^{\ell} \Pi x :^{\ell_0} A.B \qquad \Gamma \Vdash a :^{\ell_0 \vee \ell} A}{\Gamma \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}}\ \textsc{T-AppC}$$

- $\Pi x :^{\ell} A.B$ acts a little like $\Pi x : (T^{\ell}\ A).B$, so rule $\textsc{T-Abs}$ looks like rule $\textsc{SDC-Bind}$ and rule $\textsc{T-App}$ looks like rule $\textsc{SDC-Return}$.

Type system in depth

T-Abs

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

T-Pi

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

T-AppC

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

- $\Pi x :^{\ell} A.B$ acts a little like $\Pi x : (T^{\ell}\ A).B$, so rule T-Abs looks like rule SDC-Bind and rule T-App looks like rule SDC-Return.
- Important difference: x labeled with ℓ instead of $\ell_0 \vee \ell$ in rule T-Pi.

Type system in depth

$$\textsc{T-Abs}\quad \frac{\Gamma, x :^{\ell_0 \vee \ell} A \vdash b :^{\ell} B \qquad C \wedge \Gamma \vdash (\Pi x :^{\ell_0} A.B) :^{C} s}{\Gamma \vdash \lambda x :^{\ell_0} A.b :^{\ell} \Pi x :^{\ell_0} A.B}$$

$$\textsc{T-Pi}\quad \frac{\Gamma \vdash A :^{\ell} s_1 \qquad \Gamma, x :^{\ell} A \vdash B :^{\ell} s_2 \qquad \mathcal{R}(s_1, s_2, s_3)}{\Gamma \vdash \Pi x :^{\ell_0} A.B :^{\ell} s_3}$$

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

- $\Pi x :^{\ell} A.B$ acts a little like $\Pi x : (T^{\ell}\ A).B$, so rule T-Abs looks like rule SDC-Bind and rule T-App looks like rule SDC-Return.
- Important difference: x labeled with ℓ instead of $\ell_0 \vee \ell$ in rule T-Pi.
- To know result type of rule T-App is well-formed, have $C \wedge \Gamma \vdash \Pi x :^{\ell_0} A.B :^{C} s$, so label of a must be $\leq C$

Type system in depth

$$\textsc{T-Abs}\quad \frac{\Gamma, x :^{\ell_0 \vee \ell} A \vdash b :^{\ell} B \qquad C \wedge \Gamma \vdash (\Pi x :^{\ell_0} A.B) :^{C} s}{\Gamma \vdash \lambda x :^{\ell_0} A.b :^{\ell} \Pi x :^{\ell_0} A.B}$$

$$\textsc{T-Pi}\quad \frac{\Gamma \vdash A :^{\ell} s_1 \qquad \Gamma, x :^{\ell} A \vdash B :^{\ell} s_2 \qquad \mathcal{R}(s_1, s_2, s_3)}{\Gamma \vdash \Pi x :^{\ell_0} A.B :^{\ell} s_3}$$

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

- $\Pi x :^{\ell} A.B$ acts a little like $\Pi x : (T^{\ell}\ A).B$, so rule T-ABS looks like rule SDC-BIND and rule T-APP looks like rule SDC-RETURN.
- Important difference: x labeled with ℓ instead of $\ell_0 \vee \ell$ in rule T-PI.
- To know result type of rule T-APP is well-formed, have $C \wedge \Gamma \vdash \Pi x :^{\ell_0} A.B :^{C} s$, so label of a must be $\leq C$
- Resurrection ensures C is highest label on judgement.

Related Work

DDC is only type system with multiple, independent levels of irrelevance. This distinction is essential for strong Σ-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.

Related Work

DDC is only type system with multiple, independent levels of irrelevance. This distinction is essential for strong Σ-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.

Related Work

DDC is only type system with multiple, independent levels of irrelevance. This distinction is essential for strong Σ-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 Π must match use in λ.

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)*

$$\textsc{T-Var}\quad \frac{\ell_0 \leq \ell \qquad x :^{\ell_0} A \in \Gamma \qquad \ell \leq C}{\Gamma \vdash x :^{\ell} A}$$

$$\textsc{T-Pi}\quad \frac{\Gamma \vdash A :^{\ell} s_1 \qquad \Gamma, x :^{\ell} A \vdash B :^{\ell} s_2 \qquad \mathcal{R}(s_1, s_2, s_3)}{\Gamma \vdash \Pi x :^{\ell_0} A.B :^{\ell} s_3}$$

$$\textsc{T-AbsC}\quad \frac{\Gamma, x :^{\ell_0 \vee \ell} A \vdash b :^{\ell} B \qquad \Gamma \Vdash (\Pi x :^{\ell_0} A.B) :^{\top} s}{\Gamma \vdash \lambda x :^{\ell_0} A.b :^{\ell} \Pi x :^{\ell_0} A.B}$$

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

$$\textsc{T-ConvC}\quad \frac{\Gamma \vdash a :^{\ell} A \qquad |C \wedge \Gamma| \vdash A \equiv_C B \qquad \Gamma \Vdash B :^{\top} s}{\Gamma \vdash a :^{\ell} B}$$

$$\textsc{T-Type}\quad \frac{\ell \leq C \qquad \mathcal{A}(s_1, s_2)}{\Gamma \vdash s_1 :^{\ell} s_2}$$

Typing rules for DDC (continued)

$\boxed{\Gamma \Vdash a :^{\ell} A}$ *(Truncate at $\top$)*

$$\textsc{CT-Leq}\quad \frac{\Gamma \vdash a :^{\ell} A \qquad \ell \leq C}{\Gamma \Vdash a :^{\ell} A}$$

$$\textsc{CT-Top}\quad \frac{C \wedge \Gamma \vdash a :^{C} A \qquad C < \ell}{\Gamma \Vdash a :^{\ell} A}$$

Typing rules for Σ-types

T-SPair

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

T-LetPairC

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

T-Proj1

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

T-Proj2

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