Adjoint Logic:
Multiple Modalities into a Computational Framework

Junyoung Jang
junyoung.jang@mail.mcgill.ca
McGill University

Joint work with Brigitte Pientka

Adjoint Logic as a Uniform Framework

`next` Modality

Adjoint Logic as a Uniform Framework

`next` Modality

Binding-time Analysis

Model Checking

Adjoint Logic as a Uniform Framework

$\Box$ Modality

`next` Modality

Binding-time Analysis

Model Checking

Adjoint Logic as a Uniform Framework

$\Box$ Modality

Staged programming

`next` Modality

Binding-time Analysis

Model Checking

Adjoint Logic as a Uniform Framework

$\Box$ Modality

Staged programming

`next` Modality

Binding-time Analysis

Model Checking

$\bigcirc$ Modality

Adjoint Logic as a Uniform Framework

$\Box$ Modality

Staged programming

`next` Modality

Binding-time Analysis

Model Checking

$\bigcirc$ Modality

Information Flow

Reactive Programming

Adjoint Logic as a Uniform Framework

$\Box$ Modality

Staged programming

`next` Modality

Binding-time Analysis

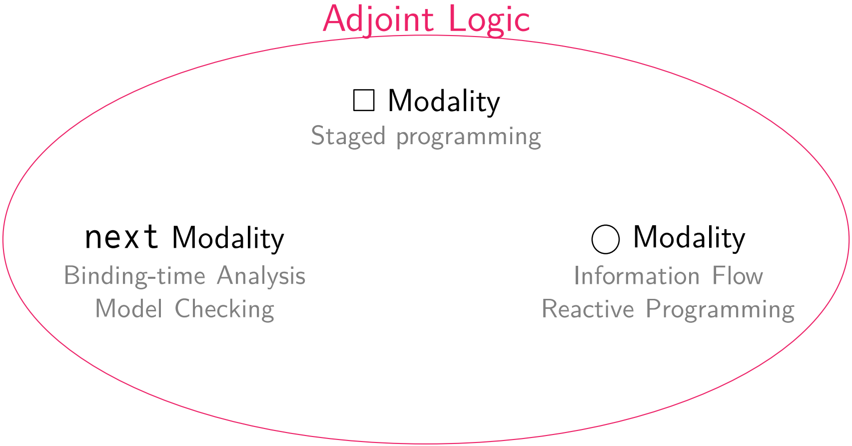
Model Checking

$\bigcirc$ Modality

Information Flow

Reactive Programming

Adjoint Logic as a Uniform Framework



Adjoint Logic

□ Modality

Staged programming

`next` Modality

Binding-time Analysis
Model Checking

◯ Modality

Information Flow
Reactive Programming

Adjoint Logic as a Uniform Framework

Adjoint Logic

$\Box$ Modality

Staged programming

`next` Modality

Binding-time Analysis

Model Checking

$\bigcirc$ Modality

Information Flow

Reactive Programming

while keeping static/dynamic semantics!

Adjoint Logic

Uniform logic with two adjoint functors $\uparrow$ (upshift)/$\downarrow$ (downshift) connecting different sublogics:

Adjoint Logic

Uniform logic with two adjoint functors ↑ (upshift)/↓ (downshift) connecting different sublogics:

- To connect linear and intuitionistic logic [Benton, 1995]

Adjoint Logic

Uniform logic with two adjoint functors $\uparrow$ (upshift)/$\downarrow$ (downshift) connecting different sublogics:

- To connect linear and intuitionistic logic [Benton, 1995]
- To connect multiple logics in more general ways[Reed, 2009, Licata and Shulman, 2016]

Adjoint Logic

Uniform logic with two adjoint functors ↑ (upshift)/↓ (downshift) connecting different sublogics:

- To connect linear and intuitionistic logic [Benton, 1995]
- To connect multiple logics in more general ways[Reed, 2009, Licata and Shulman, 2016]

Adjoint Logic

Uniform logic with two adjoint functors ↑ (upshift)/↓ (downshift) connecting different sublogics:

- To connect linear and intuitionistic logic [Benton, 1995]
- To connect multiple logics in more general ways[Reed, 2009, Licata and Shulman, 2016]

However,

- No decidable type checking

Adjoint Logic

Uniform logic with two adjoint functors $\uparrow$ (upshift)/$\downarrow$ (downshift) connecting different sublogics:

- To connect linear and intuitionistic logic [Benton, 1995]
- To connect multiple logics in more general ways[Reed, 2009, Licata and Shulman, 2016]

However,

- No decidable type checking
- No syntactic operational/reduction semantics

Adjoint Logic

Uniform logic with two adjoint functors ↑ (upshift)/↓ (downshift) connecting different sublogics:

- To connect linear and intuitionistic logic [Benton, 1995]
- To connect multiple logics in more general ways[Reed, 2009, Licata and Shulman, 2016]

However,

- No decidable type checking
- No syntactic operational/reduction semantics
- No computational interpretation for ↑/↓

Adjoint Logic in Action

Practical and uniform foundation for proof/programming about modalities with

Adjoint Logic in Action

Practical and uniform foundation for proof/programming about modalities with

- Decidable type checking

Adjoint Logic in Action

Practical and uniform foundation for proof/programming about modalities with

- Decidable type checking
- Operational/reduction semantics

Adjoint Logic in Action

Practical and uniform foundation for proof/programming about modalities with

- Decidable type checking
- Operational/reduction semantics
- Computational interpretation for $\uparrow/\downarrow$

Adjoint Logic in Action

Practical and uniform foundation for proof/programming about modalities with

- Decidable type checking
- Operational/reduction semantics
- Computational interpretation for $\uparrow$/$\downarrow$
- Safety properties

Adjoint Logic in Action

Practical and uniform foundation for proof/programming about modalities with

- Decidable type checking
- Operational/reduction semantics
- Computational interpretation for $\uparrow$/$\downarrow$
- Safety properties
- Embedding $\Box$/$\mathtt{next}$/$\bigcirc$ modalities

Adjoint Logic in Action

Practical and uniform foundation for proof/programming about modalities with

- Decidable type checking
- Operational/reduction semantics
- Computational interpretation for $\uparrow/\downarrow$
- Safety properties
- Embedding $\Box/\texttt{next}/\bigcirc$ modalities

Adjoint Logic in Action

Practical and uniform foundation for proof/programming about modalities with

- Decidable type checking
- Operational/reduction semantics
- Computational interpretation for $\uparrow/\downarrow$
- Safety properties
- Embedding $\Box/\texttt{next}/\bigcirc$ modalities while preserving static and dynamic semantics

Adjoint Logic in Action

Practical and uniform foundation for proof/programming about modalities with

- Decidable type checking
- Operational/reduction semantics
- Computational interpretation for $\uparrow/\downarrow$
- Safety properties
- Embedding $\Box/\texttt{next}/\bigcirc$ modalities
  while preserving static and dynamic semantics

“ELEVATOR”

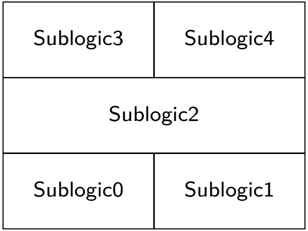
Adjoint Logic in Action

Practical and uniform foundation for proof/programming about modalities with

- Decidable type checking
- Operational/reduction semantics
- Computational interpretation for $\uparrow/\downarrow$
- Safety properties
- Embedding $\Box/\mathtt{next}/\bigcirc$ modalities
  while preserving static and dynamic semantics

“ELEVATOR”

Overview of ELEVATOR


Sublogic3

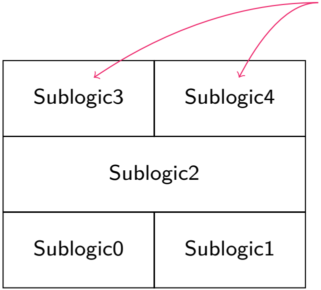
Sublogic4

Sublogic2

Sublogic0

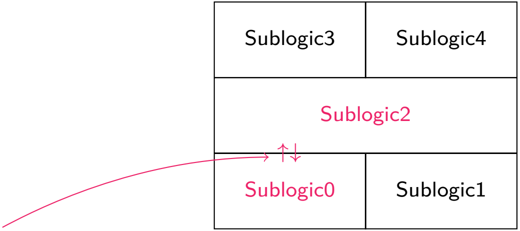
Sublogic1

Overview of ELEVATOR



There are multiple sublogics
each of which resides in one "mode"

Overview of ELEVATOR



Sublogic3

Sublogic4

Sublogic2

$\uparrow\downarrow$

Sublogic0

Sublogic1

Adjoint modalities ↑ and ↓ allow interaction between sublogics

Mode Specification

Modes are members of a preorder
Each mode controls allowed structural rules and types in its sublogic.

Mode Specification

Modes are members of a preorder
Each mode controls allowed structural rules and types in its sublogic.

For example,

u -	$\{\mathtt{Wk}, \mathtt{Co}\}, \{\uparrow\}$
l -	$\{\}, \{\downarrow, \multimap, \mathtt{Nat}\}$

u -	$\{\mathtt{Wk}, \mathtt{Co}\}, \{\uparrow, \rightarrow, \mathtt{Nat}\}$
l -	$\{\}, \{\downarrow, \multimap, \mathtt{Nat}\}$

(where $u \geq l$)

Mode Specification

Modes are members of a preorder

Each mode controls allowed structural rules and types in its sublogic.

For example,

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow\}$
l -	$\{\}$	, $\{\downarrow, \multimap, \mathtt{Nat}\}$

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow, \rightarrow, \mathtt{Nat}\}$
l -	$\{\}$	, $\{\downarrow, \multimap, \mathtt{Nat}\}$

(where $u \geq l$)

Dual intuitionistic linear logic (DILL) [Barber and Plotkin, 1996]

Mode Specification

Modes are members of a preorder
Each mode controls allowed structural rules and types in its sublogic.

For example,

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow\}$
l -	$\{\}$	, $\{\downarrow, \multimap, \mathtt{Nat}\}$

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow, \rightarrow, \mathtt{Nat}\}$
l -	$\{\}$	, $\{\downarrow, \multimap, \mathtt{Nat}\}$

(where $u \geq l$)

Linear/non-linear logic (LNL) [Benton, 1995]

Mode Specification

Modes are members of a preorder

Each mode controls allowed structural rules and types in its sublogic.

For example,

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow\}$
l -	$\{\}$	, $\{\downarrow, \multimap, \mathtt{Nat}\}$

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow, \rightarrow, \mathtt{Nat}\}$
l -	$\{\}$	, $\{\downarrow, \multimap, \mathtt{Nat}\}$

Higher, More global, More long-lasting

↑

Lower, More local, More temporary

(where $u \geq l$)

Mode Specification

Modes are members of a preorder
Each mode controls allowed structural rules and types in its sublogic.

For example,

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow\}$
l -	$\{\}$	, $\{\downarrow, \multimap, \mathtt{Nat}\}$

Weakening

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow, \rightarrow, \mathtt{Nat}\}$
l -	$\{\}$	, $\{\downarrow, \multimap, \mathtt{Nat}\}$

(where $u \geq l$)

Mode Specification

Modes are members of a preorder
Each mode controls allowed structural rules and types in its sublogic.

For example,

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow\}$
l -	$\{\}$	$, \{\downarrow, \multimap, \mathtt{Nat}\}$

Weakening

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow, \rightarrow, \mathtt{Nat}\}$
l -	$\{\}$	$, \{\downarrow, \multimap, \mathtt{Nat}\}$

(where $u \geq l$)

Mode Specification

Modes are members of a preorder
Each mode controls allowed structural rules and types in its sublogic.

For example,

Contraction

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow\}$
l -	$\{\}$	$, \{\downarrow, \multimap, \mathtt{Nat}\}$

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow, \rightarrow, \mathtt{Nat}\}$
l -	$\{\}$	$, \{\downarrow, \multimap, \mathtt{Nat}\}$

(where $u \geq l$)

Mode Specification

Modes are members of a preorder
Each mode controls allowed structural rules and types in its sublogic.

For example,

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow\}$
l -	$\{\}$	, $\{\downarrow, \multimap, \mathtt{Nat}\}$

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$,	$\{\uparrow, \rightarrow, \mathtt{Nat}\}$
l -	$\{\}$	, $\{\downarrow, \multimap, \mathtt{Nat}\}$

(where $u \geq l$)

Mode Dependency Principle

The behaviour of a sublogic of a higher mode
cannot depend on
the behaviour of a sublogic of a lower mode.

Adjoint Modalities —Computational Interpretation

Adjoint modalities ↑ and ↓ connect two comparable modes:

Adjoint Modalities —Computational Interpretation

Adjoint modalities ↑ and ↓ connect two comparable modes:

- $\uparrow_l^h S$ — A thunk at a mode h of a closed deferred expression of type S at a lower mode l

Adjoint Modalities —Computational Interpretation

Adjoint modalities $\uparrow$ and $\downarrow$ connect two comparable modes:

- $\uparrow_l^h S$ — A thunk at a mode h of a closed deferred expression of type S at a lower mode l
- $\downarrow_l^h S$ — A pointer at a mode l to a stored value of type S at a higher mode h

Computational Interpretation of $\uparrow_l^h S$

Computational Interpretation of $\uparrow_l^h S$

- $\uparrow_l^h S$ — A thunk at a mode h of a closed deferred expression of type S at a lower mode l

Computational Interpretation of $\uparrow_l^h S$

- $\mathrm{thunk}_l^h\ (L) : \uparrow_l^h S$ — A thunk at a mode h of a closed deferred expression L of type S at a lower mode l

Computational Interpretation of $\uparrow_l^h S$

- $\text{thunk}_l^h\ (L) : \uparrow_l^h S$ — A thunk at a mode h of a closed deferred expression L of type S at a lower mode l
- $\text{force}_l^h\ (L) : S$ — compose/execute the expression in the thunk $L : \uparrow_l^h S$

Computational Interpretation of $\downarrow_{l}^{h} S$

Computational Interpretation of $\downarrow_l^h S$

- $\downarrow_l^h S$ — A pointer at a mode l to a stored value of type S at a higher mode h

Computational Interpretation of $\downarrow_l^h S$

- $\mathrm{store}_l^h(L) : \downarrow_l^h S$ — A pointer at a mode l to a stored value of L of type S at a higher mode h

Computational Interpretation of $\downarrow_l^h S$

- $\text{store}_l^h\ (L) : \downarrow_l^h S$ — A pointer at a mode l to a stored value of L of type S at a higher mode h
- $\text{load}_l^h\ (x) = L \text{ in } M$ — load the value of type S referred by the pointer $L : \downarrow_l^h S$ into x and continue with M

$\lambda^{\Box}$ —A foundation for staged programming

$\lambda^{\Box}$ [Davies and Pfenning, 2001]

- $\Box A$ describes a code fragment of type A

$\lambda^{\square}$ —A foundation for staged programming

$\lambda^{\square}$ [Davies and Pfenning, 2001]

- $\square A$ describes a code fragment of type A

c - $\{\mathtt{Wk}, \mathtt{Co}\}$, $\{\uparrow\}$
p - $\{\mathtt{Wk}, \mathtt{Co}\}$, $\{\downarrow, \rightarrow, \mathtt{Nat}\}$

Staging the Power Function

c - {Wk, Co} , $\{\uparrow\}$

p - {Wk, Co} , $\{\downarrow, \to,$ Nat$\}$

```
pow :$^p$ Nat → $\downarrow^c_p \uparrow^c_p$(Nat → Nat)
pow 0       =
  store$^c_p$ (thunk$^c_p$ (fun x → 1))
pow (suc n) =
  load$^c_p$ P = pow n in
  store$^c_p$ (thunk$^c_p$ (fun x → x * ((force$^c_p$ P) x))
```

Staging the Power Function

c -	{Wk, Co} , {↑}
p -	{Wk, Co} , {↓, →, Nat}

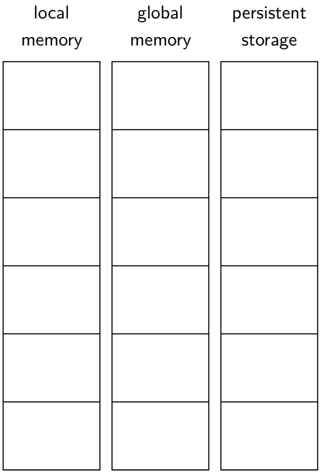
Encoding of □

```
pow :^p Nat → ↓_p^c ↑_p^c (Nat → Nat)
pow 0        =
  store_p^c (thunk_p^c (fun x → 1))
pow (suc n) =
  load_p^c P = pow n in
  store_p^c (thunk_p^c (fun x → x * ((force_p^c P) x))
```

Staging the Power Function

c -	{Wk, Co} , {↑}
p -	{Wk, Co} , {↓, →, Nat}

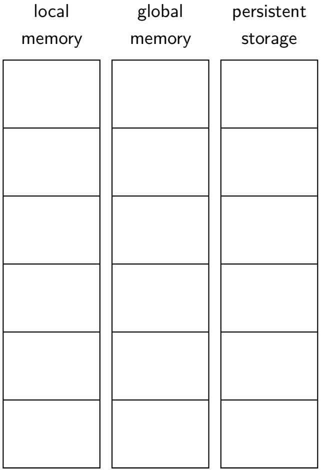
```
pow :^p Nat → ↓_p^c ↑_p^c(Nat → Nat)
pow 0       =
  store_p^c (thunk_p^c (fun x → 1))
pow (suc n) =
  load_p^c P = pow n in
  store_p^c (thunk_p^c (fun x → x * ((force_p^c P) x))
```



Staging the Power Function

c - {Wk, Co} , {↑}
p - {Wk, Co} , {↓, →, Nat}

```
pow :^p Nat → ↓^c_p ↑^c_p(Nat → Nat)
pow 0        =
  store^c_p (thunk^c_p (fun x → 1))
pow (suc n) =
  load^c_p P = pow n in
  store^c_p (thunk^c_p (fun x → x * ((force^c_p P) x))
```

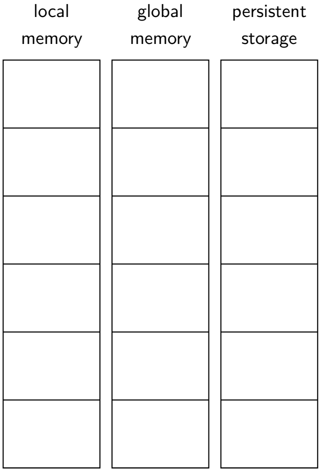


Staging the Power Function

c - {Wk, Co} , {↑}
p - {Wk, Co} , {↓, →, Nat}

Construct a thunk of type $\uparrow^c_p$(**Nat**→**Nat**)
for **fun** x → 1 (for x^0)

```
pow :^p Nat → $\downarrow^c_p \uparrow^c_p$(Nat → Nat)
pow 0        =
  store$^c_p$ (thunk$^c_p$ (fun x → 1))
pow (suc n) =
  load$^c_p$ P = pow n in
  store$^c_p$ (thunk$^c_p$ (fun x → x * ((force$^c_p$ P) x))
```



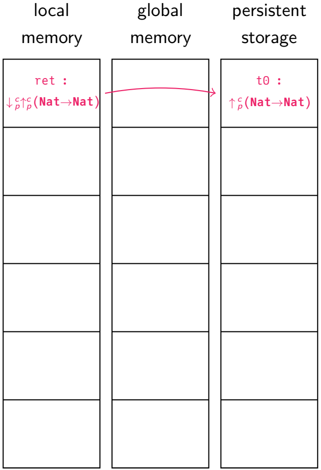
Staging the Power Function

c - {Wk, Co} , {↑}

p - {Wk, Co} , {↓, →, Nat}

Store it into t0 and return a pointer (ret) to it

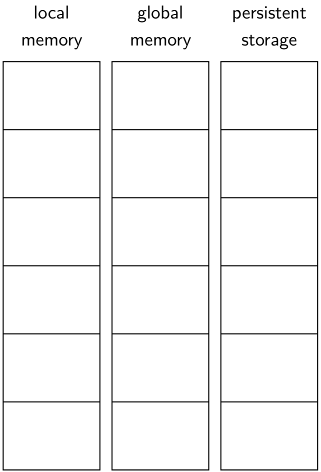
```
pow :ᵖ Nat → ↓ᶜₚ ↑ᶜₚ(Nat → Nat)
pow 0        =
  storeᶜₚ (thunkᶜₚ (fun x → 1))
pow (suc n) =
  loadᶜₚ P = pow n in
  storeᶜₚ (thunkᶜₚ (fun x → x * ((forceᶜₚ P) x))
```



Staging the Power Function

c -	{Wk, Co} , {↑}
p -	{Wk, Co} , {↓, →, Nat}

```
pow :^p Nat → ↓^c_p ↑^c_p(Nat → Nat)
pow 0         =
  store^c_p (thunk^c_p (fun x → 1))
pow (suc n) =
  load^c_p P = pow n in
  store^c_p (thunk^c_p (fun x → x * ((force^c_p P) x))
```


local memory

global memory

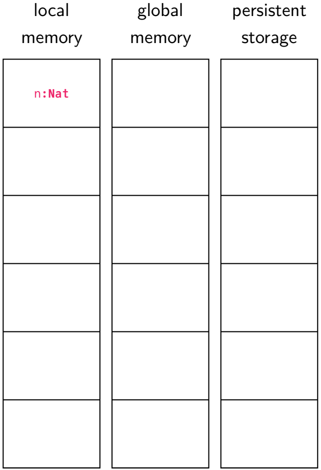
persistent storage

Staging the Power Function

c - {Wk, Co} , {↑}

p - {Wk, Co} , {↓, →, Nat}

```
pow :^p Nat → $\downarrow^c_p \uparrow^c_p$(Nat → Nat)
pow 0        =
  $\text{store}^c_p$ ($\text{thunk}^c_p$ (fun x → 1))
pow (suc n) =
  $\text{load}^c_p$ P = pow n in
  $\text{store}^c_p$ ($\text{thunk}^c_p$ (fun x → x * (($\text{force}^c_p$ P) x))
```

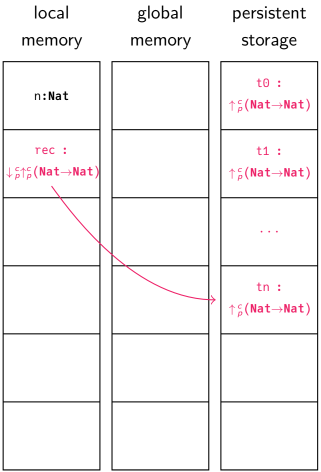


Staging the Power Function

c - {Wk, Co} , {↑}
p - {Wk, Co} , {↓, →, Nat}

Recursively build a thunk for x^n
and get a pointer (rec) to it

```
pow :^p Nat → ↓^c_p ↑^c_p(Nat → Nat)
pow 0       =
  store^c_p (thunk^c_p (fun x → 1))
pow (suc n) =
  load^c_p P = pow n in
  store^c_p (thunk^c_p (fun x → x * ((force^c_p P) x))
```

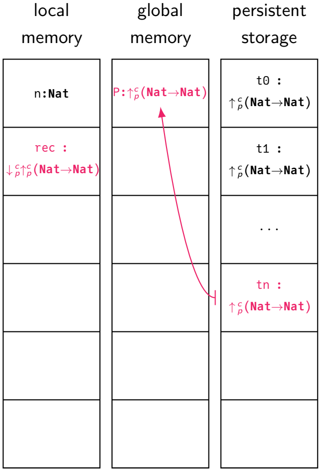


Staging the Power Function

c -	{Wk, Co} , {↑}
p -	{Wk, Co} , {↓, →, Nat}

Load the thunk referred by the pointer

```
pow :^p Nat → ↓^c_p ↑^c_p(Nat → Nat)
pow 0       =
  store^c_p (thunk^c_p (fun x → 1))
pow (suc n) =
  load^c_p P = pow n in
  store^c_p (thunk^c_p (fun x → x * ((force^c_p P) x)))
```



local memory	global memory	persistent storage
n:Nat	P:$\uparrow^c_p$(Nat→Nat)	t0 : $\uparrow^c_p$(Nat→Nat)
rec : $\downarrow^c_p\uparrow^c_p$(Nat→Nat)		t1 : $\uparrow^c_p$(Nat→Nat)
		. . .
		tn : $\uparrow^c_p$(Nat→Nat)

Staging the Power Function

c -	{Wk, Co} , {↑}
p -	{Wk, Co} , {↓, →, Nat}

Splice P into a bigger thunk

```
pow :^p Nat → ↓^c_p ↑^c_p (Nat → Nat)
pow 0        =
  store^c_p (thunk^c_p (fun x → 1))
pow (suc n) =
  load^c_p P = pow n in
  store^c_p (thunk^c_p (fun x → x * ((force^c_p P) x))
```

local memory	global memory	persistent storage
n:Nat	P: $\uparrow^c_p$(Nat→Nat)	t0 : $\uparrow^c_p$(Nat→Nat)
rec : $\downarrow^c_p\uparrow^c_p$(Nat→Nat)		t1 : $\uparrow^c_p$(Nat→Nat)
		...
		tn : $\uparrow^c_p$(Nat→Nat)

Staging the Power Function

c -	{Wk, Co} , {↑}
p -	{Wk, Co} , {↓, →, Nat}

```
pow :$^p$ Nat → $\downarrow^c_p \uparrow^c_p$(Nat → Nat)
pow 0         =
  store$^c_p$ (thunk$^c_p$ (fun x → 1))
pow (suc n) =
  load$^c_p$ P = pow n in
  store$^c_p$ (thunk$^c_p$ (fun x → x * ((force$^c_p$ P) x)))
```

Construct a thunk for x^{n+1}

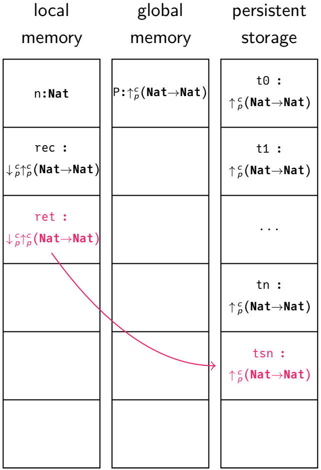
local memory	global memory	persistent storage
n:Nat	P:$\uparrow^c_p$(Nat→Nat)	t0 : $\uparrow^c_p$(Nat→Nat)
rec : $\downarrow^c_p\uparrow^c_p$(Nat→Nat)		t1 : $\uparrow^c_p$(Nat→Nat)
		...
		tn : $\uparrow^c_p$(Nat→Nat)

Staging the Power Function

c -	{Wk, Co} , {↑}
p -	{Wk, Co} , {↓, →, Nat}

Store it into tsn and get a pointer (ret) to it

```
pow $:^{p}$ Nat → $\downarrow^{c}_{p}\uparrow^{c}_{p}$(Nat → Nat)
pow 0        =
  store$^{c}_{p}$ (thunk$^{c}_{p}$ (fun x → 1))
pow (suc n) =
  load$^{c}_{p}$ P = pow n in
  store$^{c}_{p}$ (thunk$^{c}_{p}$ (fun x → x * ((force$^{c}_{p}$ P) x))
```



local memory	global memory	persistent storage
n:Nat	P:$\uparrow^{c}_{p}$(Nat→Nat)	t0 : $\uparrow^{c}_{p}$(Nat→Nat)
rec : $\downarrow^{c}_{p}\uparrow^{c}_{p}$(Nat→Nat)		t1 : $\uparrow^{c}_{p}$(Nat→Nat)
ret : $\downarrow^{c}_{p}\uparrow^{c}_{p}$(Nat→Nat)		. . .
		tn : $\uparrow^{c}_{p}$(Nat→Nat)
		tsn : $\uparrow^{c}_{p}$(Nat→Nat)

Linear Calculus with !

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$	, $\{\uparrow\}$
l -	$\{\}$	, $\{\downarrow, \multimap, \mathtt{Nat}\}$

Working with a Protocol

u -	$\{\texttt{Wk}, \texttt{Co}\}$	$, \{\uparrow\}$
l -	$\{\}$	$, \{\downarrow, \multimap, \texttt{Nat}\}$

- Linear types can encode session types

Working with a Protocol

u -	$\{\mathtt{Wk}, \mathtt{Co}\}$	$, \{\uparrow\}$
l -	$\{\}$	$, \{\downarrow, \multimap, \mathtt{Nat}\}$

- Linear types can encode session types
- Unrestricted types can encode functional program

Linear Calculus with $\Box$ and ! – Protocol with Remote Execution

u - $\{\texttt{Wk}, \texttt{Co}\}, \{\uparrow\}$	c - $\{\}, \{\uparrow\}$
l - $\{\}, \{\downarrow, \multimap, \texttt{Nat}\}$	

Lambda Calculus with $\bigcirc$ — Dead Code Analysis

p - $\{\mathtt{Wk}, \mathtt{Co}\}$, $\{\uparrow, \rightarrow, \mathtt{Nat}\}$
s - $\{\mathtt{Wk}, \mathtt{Co}\}$, $\{\downarrow\}$

$$\bigcirc A = \uparrow^{p}_{s} \downarrow^{p}_{s} A$$

Lambda Calculus with `next` — Binding Time Analysis

t_0 - $\{\texttt{Wk}, \texttt{Co}\}$, $\{\rightarrow, \texttt{Nat}\}$
t_1 - $\{\texttt{Wk}, \texttt{Co}\}$, $\{\downarrow, \rightarrow, \texttt{Nat}\}$
$\ldots$

$$\texttt{next}\ A = \downarrow^{t_n}_{t_{n+1}} A$$

Typing Judgement

$$\Gamma \vdash^m L : S$$

Typing Judgement

Current mode of type checking

$$\Gamma \vdash^{m} L : S$$

Typing Judgement

$x_1 :^{n_1} T_1, x_2 :^{n_2} T_2, \ldots$ where $n_i \geq m$

$$\Gamma \vdash^{m} L : S$$

Typing Rules for Adjoint Modalities

$$\frac{\Gamma \vdash^{l} L : S}{\Gamma \vdash^{m} \mathsf{thunk}_{l}^{m}\ (L) : \uparrow_{l}^{m} S} \qquad \frac{\Gamma|^{h} \vdash^{h} L : \uparrow_{m}^{h} S}{\Gamma \vdash^{m} \mathsf{force}_{m}^{h}\ (L) : S}$$

$$\frac{\Gamma|^{h} \vdash^{h} L : S}{\Gamma \vdash^{m} \mathsf{store}_{m}^{h}\ (L) : \downarrow_{m}^{h} S}$$

$$\frac{m \geq^{\mathcal{M}} n \qquad \Gamma' \vdash^{m} L : \downarrow_{m}^{h} T \qquad \Gamma'', x{:}^{h} T \vdash^{n} M : S}{\Gamma'; \Gamma'' \vdash^{n} \mathsf{load}_{m}^{h}\ (x) = L \ \mathsf{in}\ M : S}\ \mathsf{E}{\downarrow}$$

Typing Rules for Adjoint Modalities

$\Gamma|^h = \Gamma'$ cuts all entries in Γ not higher than h

$$\frac{\Gamma \vdash^l L : S}{\Gamma \vdash^m \mathsf{thunk}_l^m\ (L) : \uparrow_l^m S} \qquad \frac{\Gamma|^h \vdash^h L : \uparrow_m^h S}{\Gamma \vdash^m \mathsf{force}_m^h\ (L) : S}$$

$$\frac{\Gamma|^h \vdash^h L : S}{\Gamma \vdash^m \mathsf{store}_m^h\ (L) : \downarrow_m^h S}$$

$$\frac{m \geq^{\mathcal{M}} n \qquad \Gamma' \vdash^m L : \downarrow_m^h T \qquad \Gamma'', x{:}^h T \vdash^n M : S}{\Gamma'; \Gamma'' \vdash^n \mathsf{load}_m^h\ (x) = L \text{ in } M : S}\ \mathsf{E}{\downarrow}$$

Typing Rules for Adjoint Modalities

$\Gamma' ; \Gamma'' = \Gamma$ splits Γ into two contexts

$$\frac{\Gamma \vdash^{l} L : S}{\Gamma \vdash^{m} \mathsf{thunk}_{l}^{m}\ (L) : {\uparrow}_{l}^{m}\ S} \qquad \frac{\Gamma|^{h} \vdash^{h} L : {\uparrow}_{m}^{h}\ S}{\Gamma \vdash^{m} \mathsf{force}_{m}^{h}\ (L) : S}$$

$$\frac{\Gamma|^{h} \vdash^{h} L : S}{\Gamma \vdash^{m} \mathsf{store}_{m}^{h}\ (L) : {\downarrow}_{m}^{h}\ S}$$

$$\frac{m \geq^{\mathcal{M}} n \qquad \Gamma' \vdash^{m} L : {\downarrow}_{m}^{h}\ T \qquad \Gamma'', x{:}^{h} T \vdash^{n} M : S}{\Gamma' ; \Gamma'' \vdash^{n} \mathsf{load}_{m}^{h}\ (x) = L \ \mathsf{in}\ M : S}\ \mathsf{E}{\downarrow}$$

Operational Semantics

$$L \longrightarrow L'$$

$$L \longrightarrow^{m\leq} L'$$

Operational Semantics

$L \longrightarrow L'$ — reduction of redex

$L \longrightarrow^{m\leq} L'$

Operational Semantics

$L \longrightarrow L'$ — reduction of redex

$L \longrightarrow^{m\leq} L'$ — reduction of redex at mode $\geq m$ in deferred expression

Type Safety

Theorem (Type Preservation)

For $\Gamma \vdash^n L : S$,

1. *If* $L \longrightarrow L'$, *then* $\Gamma \vdash^n L' : S$
2. *For any mode* m, *if* $L \longrightarrow^{m\leq} L'$, *then* $\Gamma \vdash^n L' : S$

Type Safety

Theorem (Type Preservation)

For $\Gamma \vdash^n L : S$,

1. *If* $L \longrightarrow L'$, *then* $\Gamma \vdash^n L' : S$
2. *For any mode m, if* $L \longrightarrow^{m\leq} L'$, *then* $\Gamma \vdash^n L' : S$

Theorem (Progress)

For $\Gamma \vdash^n L : S$,

1. *Either* L *is a weak normal form or there exists* L' *such that* $L \longrightarrow L'$
2. *For any mode m, either* L *is a canonical form or there exists* L' *such that* $L \longrightarrow^{m\leq} L'$

Dynamic Behaviour of Calculi

Theorem (Complete and Sound Translation)

There is an ELEVATOR *instance that keeps the well-typedness of the* $\lambda^{\Box}$ *or DILL*

Theorem (Bisimulation)

There is an ELEVATOR *instance that keeps the same dynamic semantics of the* $\lambda^{\Box}$ *or DILL*

Dynamic Behaviour of Calculi

Theorem (Complete and Sound Translation)

There is an ELEVATOR *instance that keeps the well-typedness of the* $\lambda^{\Box}$ *or DILL*

Theorem (Bisimulation)

There is an ELEVATOR *instance that keeps the same dynamic semantics of the* $\lambda^{\Box}$ *or DILL*

In other words, ELEVATOR can be used as a compilation target from those systems.

Related Work

- Fukuda and Yoshimizu [2019] describe a linear calculus with $!$ and $\Box$ modality, but does not support other substructural calculi nor $\bigcirc$ modality.

Related Work

- Fukuda and Yoshimizu [2019] describe a linear calculus with ! and $\Box$ modality, but does not support other substructural calculi nor $\bigcirc$ modality.
- Atkey [2018], Orchard et al. [2019], and Choudhury et al. [2021] present substructural calculi but cannot describe $\Box$ modality or $\bigcirc$ modality.

Related Work

- Fukuda and Yoshimizu [2019] describe a linear calculus with ! and $\Box$ modality, but does not support other substructural calculi nor $\bigcirc$ modality.
- Atkey [2018], Orchard et al. [2019], and Choudhury et al. [2021] present substructural calculi but cannot describe $\Box$ modality or $\bigcirc$ modality.
- Abel and Bernardy [2020] provide a substructural calculus with modalities, but cannot describe $\Box$ modality with a deferred expression, which is required for staged metaprogramming, nor some combinations of multiple $\bigcirc$ modalities.

Summary of Contribution

- Static and dynamic semantics for ELEVATOR

Summary of Contribution

- Static and dynamic semantics for ELEVATOR
- Type safety of ELEVATOR

Summary of Contribution

- Static and dynamic semantics for ELEVATOR
- Type safety of ELEVATOR
- Mode safety of ELEVATOR, a new concept of safety in a multi-mode system (which corresponds to non-interference for an information flow system)

Summary of Contribution

- Static and dynamic semantics for ELEVATOR
- Type safety of ELEVATOR
- Mode safety of ELEVATOR, a new concept of safety in a multi-mode system (which corresponds to non-interference for an information flow system)
- Preserving static/dynamic semantics of notable modal calculi ($\lambda^{\Box}$ and DILL)

Summary of Contribution

- Static and dynamic semantics for ELEVATOR
- Type safety of ELEVATOR
- Mode safety of ELEVATOR, a new concept of safety in a multi-mode system (which corresponds to non-interference for an information flow system)
- Preserving static/dynamic semantics of notable modal calculi ($\lambda^{\square}$ and DILL)
- Algorithmic typing of ELEVATOR

Summary of Contribution

- Static and dynamic semantics for ELEVATOR
- Type safety of ELEVATOR
- Mode safety of ELEVATOR, a new concept of safety in a multi-mode system (which corresponds to non-interference for an information flow system)
- Preserving static/dynamic semantics of notable modal calculi ($\lambda^{\Box}$ and DILL)
- Algorithmic typing of ELEVATOR
- Implementation of type checker and interpreter for ELEVATOR

Summary of Contribution

- Static and dynamic semantics for ELEVATOR
- Type safety of ELEVATOR
- Mode safety of ELEVATOR, a new concept of safety in a multi-mode system (which corresponds to non-interference for an information flow system)
- Preserving static/dynamic semantics of notable modal calculi ($\lambda^{\Box}$ and DILL)
- Algorithmic typing of ELEVATOR
- Implementation of type checker and interpreter for ELEVATOR
- Mechanization of proofs in Agda

Conclusion

- We now have a practical and uniform foundation to integrate substructural calculi and a wide-range of modalities

Conclusion

- We now have a practical and uniform foundation to integrate substructural calculi and a wide-range of modalities
- Extending ELEVATOR to System F
  — it can be a core calculus for a real-world programming languages

Conclusion

- We now have a practical and uniform foundation to integrate substructural calculi and a wide-range of modalities
- Extending ELEVATOR to System F
  — it can be a core calculus for a real-world programming languages
- Extending ELEVATOR to dependent types
  — it can be a core theory for a proof assistant with tactics/effect

Conclusion

- We now have a practical and uniform foundation to integrate substructural calculi and a wide-range of modalities
- Extending Elevator to System F
  — it can be a core calculus for a real-world programming languages
- Extending Elevator to dependent types
  — it can be a core theory for a proof assistant with tactics/effect
- Extending Elevator to different (e.g. imperative) base languages
  — it can be a core calculus for cross-language interactions

Algorithmic Typing Judgement

$$\Gamma \vdash^{m} L : S / \Gamma'$$

Algorithmic Typing Judgement

$$\Gamma \vdash^{m} L : S / \Gamma'$$

Algorithmic Typing Rules for Adjoint Modalities

$$\frac{\Gamma \vdash^{l} L : S/\Gamma'}{\Gamma \vdash^{m} \mathsf{thunk}_{l}^{m}\ (L) : {\uparrow}_{l}^{m}\ S/\Gamma'} \qquad \frac{\Gamma|^{h} \vdash^{h} L : {\uparrow}_{m}^{h}\ S/\Gamma''}{\Gamma \vdash^{m} \mathsf{force}_{m}^{h}\ (L) : S/\Gamma''}$$

$$\frac{\Gamma|^{h} \vdash^{h} L : S/\Gamma''}{\Gamma \vdash^{m} \mathsf{store}_{m}^{h}\ (L) : {\downarrow}_{m}^{h}\ S/\Gamma''}$$

$$\frac{m \geq^{\mathcal{M}} n \qquad \Gamma \vdash^{m} L : {\downarrow}_{m}^{h}\ T/\Gamma_1' \qquad \Gamma, x{:}^{h} T \vdash^{n} M : S/\Gamma_2'}{\Gamma \vdash^{n} \mathsf{load}_{m}^{h}\ (x) = L \ \mathsf{in}\ M : S/(\Gamma_1'; (\Gamma_2' \backslash x{:}^{h} T))}\ \mathsf{E}{\downarrow}$$

Algorithmic Typing Rules for Adjoint Modalities

$$\frac{\Gamma \vdash^{l} L : S/\Gamma'}{\Gamma \vdash^{m} \mathsf{thunk}_{l}^{m}\ (L) : \uparrow_{l}^{m} S/\Gamma'} \qquad \frac{\Gamma|^{h} \vdash^{h} L : \uparrow_{m}^{h} S/\Gamma''}{\Gamma \vdash^{m} \mathsf{force}_{m}^{h}\ (L) : S/\Gamma''}$$

$$\frac{\Gamma|^{h} \vdash^{h} L : S/\Gamma''}{\Gamma \vdash^{m} \mathsf{store}_{m}^{h}\ (L) : \downarrow_{m}^{h} S/\Gamma''}$$

$$\frac{m \geq^{\mathcal{M}} n \qquad \Gamma \vdash^{m} L : \downarrow_{m}^{h} T/\Gamma_1' \qquad \Gamma, x{:}^{h} T \vdash^{n} M : S/\Gamma_2'}{\Gamma \vdash^{n} \mathsf{load}_{m}^{h}\ (x) = L \mathsf{\ in\ } M : S/(\Gamma_1'; (\Gamma_2' \backslash x{:}^{h} T))}\ \mathsf{E}{\downarrow}$$

Equivalence between Two Typings

Theorem (Equivalence between two typings)

There exists Γ' *such that* $\Gamma \vdash^m L : S/\Gamma'$ *and for any* $x{:}^n T \in \Gamma$, $\Gamma' \backslash x{:}^n T \Gamma''$ *is true for some* Γ'' *if and only if* $\Gamma \vdash^m L : S$.