Multi-modal Programming with Resource Guarantees

Junyoung/“Clare” Jang

McGill University

Multi-modal Programming with Resource Guarantees
Junyoung/“Clare” Jang
McGill University

Hello, I’m Junyoung Jang, or Clare Jang from the CompLogic group, McGill University. Today, I’m gonna present you my research work on “Multi-modal programming with resource guarantees.”

Useful Modalities for Various Applications

Useful Modalities for Various Applications

First, why do we care about multiple modalities? Because we have multiple examples of useful modalities. For example, we use a modality for metaprogramming, to answer "how to characterize code fragments". Likewise, for resource availability, we use a modality to answer "how many resources are available". We use a modality for resource privacy as well, to tell whether resource access is permitted or not.

Useful Modalities for Various Applications

Metaprogramming

Useful Modalities for Various Applications
Metaprogramming

First, why do we care about multiple modalities? Because we have multiple examples of useful modalities. For example, we use a modality for metaprogramming, to answer "how to characterize code fragments". Likewise, for resource availability, we use a modality to answer "how many resources are available". We use a modality for resource privacy as well, to tell whether resource access is permitted or not.

Useful Modalities for Various Applications

Metaprogramming

How to characterize code fragments?

Useful Modalities for Various Applications

Metaprogramming

How to characterize code fragments?

First, why do we care about multiple modalities? Because we have multiple examples of useful modalities. For example, we use a modality for metaprogramming, to answer "how to characterize code fragments". Likewise, for resource availability, we use a modality to answer "how many resources are available". We use a modality for resource privacy as well, to tell whether resource access is permitted or not.

Useful Modalities for Various Applications

Metaprogramming

How to characterize code fragments?

Resource Availability

Useful Modalities for Various Applications
Metaprogramming
How to characterize code fragments?
Resource Availability

First, why do we care about multiple modalities? Because we have multiple examples of useful modalities. For example, we use a modality for metaprogramming, to answer "how to characterize code fragments". Likewise, for resource availability, we use a modality to answer "how many resources are available". We use a modality for resource privacy as well, to tell whether resource access is permitted or not.

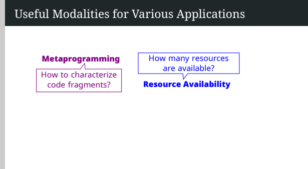
Useful Modalities for Various Applications

Metaprogramming

How to characterize code fragments?

How many resources are available?

Resource Availability



First, why do we care about multiple modalities? Because we have multiple examples of useful modalities. For example, we use a modality for metaprogramming, to answer "how to characterize code fragments". Likewise, for resource availability, we use a modality to answer "how many resources are available". We use a modality for resource privacy as well, to tell whether resource access is permitted or not.

Useful Modalities for Various Applications

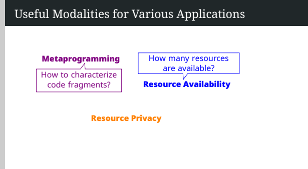
Metaprogramming

How to characterize code fragments?

How many resources are available?

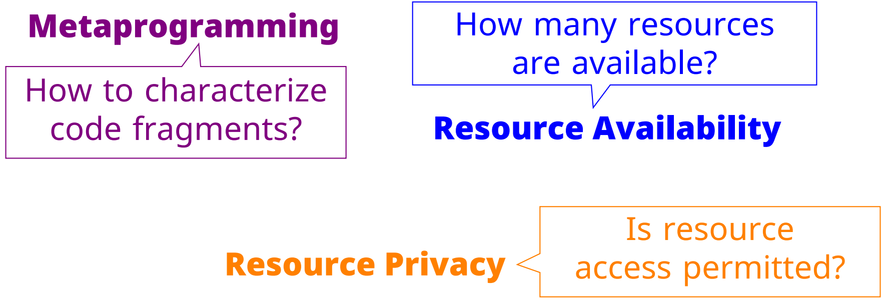
Resource Availability

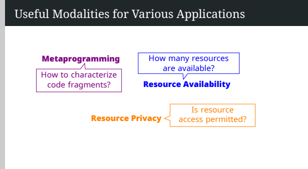
Resource Privacy



First, why do we care about multiple modalities? Because we have multiple examples of useful modalities. For example, we use a modality for metaprogramming, to answer "how to characterize code fragments". Likewise, for resource availability, we use a modality to answer "how many resources are available". We use a modality for resource privacy as well, to tell whether resource access is permitted or not.

Useful Modalities for Various Applications





First, why do we care about multiple modalities? Because we have multiple examples of useful modalities. For example, we use a modality for metaprogramming, to answer "how to characterize code fragments". Likewise, for resource availability, we use a modality to answer "how many resources are available". We use a modality for resource privacy as well, to tell whether resource access is permitted or not.

Modalities in Essence

Metaprogramming

Resource Availability

Resource Privacy

Modalities in Essence
Metaprogramming
Resource Availability
Resource Privacy

However, they all share one common aspect, arising from the very concept of modalities. Metaprogramming has a distinction between code and program, where code fragments are intensionally analyzed. In the case of resource availability, we distinguish linear resources from unrestricted resources, while allowing structural rules only for unrestricted resources. Likewise, we again put a distinction between two things, private secure data and public data, for resource privacy. In other words, they all have some “distinctions” between two modes, with some extra features such as intensional analysis or restricted structural rules. This observation allows us to capture all these modalities in a single uniform framework.

Modalities in Essence

Metaprogramming
Code/Program Distinction
with Intensional Code Fragments

Resource Availability

Resource Privacy

Modalities in Essence
Metaprogramming
Code/Program Distinction
with Intensional Code Fragments
Resource Availability
Resource Privacy

However, they all share one common aspect, arising from the very concept of modalities. Metaprogramming has a distinction between code and program, where code fragments are intensionally analyzed. In the case of resource availability, we distinguish linear resources from unrestricted resources, while allowing structural rules only for unrestricted resources. Likewise, we again put a distinction between two things, private secure data and public data, for resource privacy. In other words, they all have some "distinctions" between two modes, with some extra features such as intensional analysis or restricted structural rules. This observation allows us to capture all these modalities in a single uniform framework.

Modalities in Essence

Metaprogramming
Code/Program Distinction
with Intensional Code Fragments

Resource Availability
Linear/Unrestricted Distinction
with Restricted Structural Rules

Resource Privacy

Modalities in Essence
Metaprogramming
Code/Program Distinction
with Intensional Code Fragments
Resource Availability
Linear/Unrestricted Distinction
with Restricted Structural Rules
Resource Privacy

However, they all share one common aspect, arising from the very concept of modalities. Metaprogramming has a distinction between code and program, where code fragments are intensionally analyzed. In the case of resource availability, we distinguish linear resources from unrestricted resources, while allowing structural rules only for unrestricted resources. Likewise, we again put a distinction between two things, private secure data and public data, for resource privacy. In other words, they all have some “distinctions” between two modes, with some extra features such as intensional analysis or restricted structural rules. This observation allows us to capture all these modalities in a single uniform framework.

Modalities in Essence

Metaprogramming
Code/Program Distinction
with Intensional Code Fragments

Resource Availability
Linear/Unrestricted Distinction
with Restricted Structural Rules

Resource Privacy
Private/Public Distinction

Modalities in Essence
Metaprogramming
Code/Program Distinction
with Intensional Code Fragments
Resource Availability
Linear/Unrestricted Distinction
with Restricted Structural Rules
Resource Privacy
Private/Public Distinction

However, they all share one common aspect, arising from the very concept of modalities. Metaprogramming has a distinction between code and program, where code fragments are intensionally analyzed. In the case of resource availability, we distinguish linear resources from unrestricted resources, while allowing structural rules only for unrestricted resources. Likewise, we again put a distinction between two things, private secure data and public data, for resource privacy. In other words, they all have some “distinctions” between two modes, with some extra features such as intensional analysis or restricted structural rules. This observation allows us to capture all these modalities in a single uniform framework.

Modalities in Essence

Metaprogramming
Distinction
with Intensional Code Fragments

Resource Availability
Distinction
with Restricted Structural Rules

Resource Privacy
Distinction

Modalities in Essence
Metaprogramming
Distinction
with Intensional Code Fragments
Resource Availability
Distinction
with Restricted Structural Rules
Resource Privacy
Distinction

However, they all share one common aspect, arising from the very concept of modalities. Metaprogramming has a distinction between code and program, where code fragments are intensionally analyzed. In the case of resource availability, we distinguish linear resources from unrestricted resources, while allowing structural rules only for unrestricted resources. Likewise, we again put a distinction between two things, private secure data and public data, for resource privacy. In other words, they all have some "distinctions" between two modes, with some extra features such as intensional analysis or restricted structural rules. This observation allows us to capture all these modalities in a single uniform framework.

Modalities in Essence

Single Uniform Framework

Metaprogramming
Distinction
with Intensional Code Fragments

Resource Availability
Distinction
with Restricted Structural Rules

Resource Privacy
Distinction

Modalities in Essence
Single Uniform Framework
Metaprogramming
Distinction
with Intensional Code Fragments
Resource Availability
Distinction
with Restricted Structural Rules
Resource Privacy
Distinction

However, they all share one common aspect, arising from the very concept of modalities. Metaprogramming has a distinction between code and program, where code fragments are intensionally analyzed. In the case of resource availability, we distinguish linear resources from unrestricted resources, while allowing structural rules only for unrestricted resources. Likewise, we again put a distinction between two things, private secure data and public data, for resource privacy. In other words, they all have some “distinctions” between two modes, with some extra features such as intensional analysis or restricted structural rules. This observation allows us to capture all these modalities in a single uniform framework.

Modalities for Distinction

For any two distinguished modes m_1 and m_2,

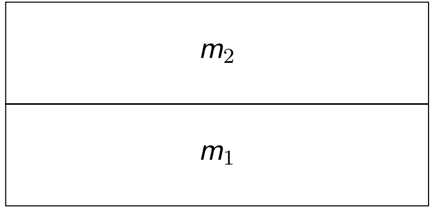
m_2
m_1

Modalities for Distinction
For any two distinguished modes m_1 and m_2,
m_2
m_1

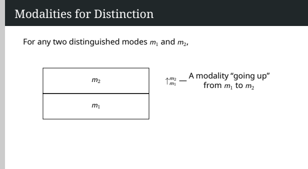
Basically, when we have two distinguished modes, m_1 and m_2, we use two modalities to connect them: Going-up modality, and going-down modality. This can be applied to any of the previous modality examples. For metaprogramming, we can put code as an upper mode and program as a lower mode. For resource availability, a mode for unrestricted resources goes on top of the linear program. Resource privacy also has a similar structure for modes. But why do we care the arrangement of modes? Why do we have two different modalities that “going-up” and “going-down”, instead of one modality for “moving-from-one-to-the-other”? This is because these two modalities has different meanings.

Modalities for Distinction

For any two distinguished modes m_1 and m_2,



$\uparrow^{m_2}_{m_1}$ — A modality "going up" from m_1 to m_2



Basically, when we have two distinguished modes, m_1 and m_2, we use two modalities to connect them: Going-up modality, and going-down modality. This can be applied to any of the previous modality examples. For metaprogramming, we can put code as an upper mode and program as a lower mode. For resource availability, a mode for unrestricted resources goes on top of the linear program. Resource privacy also has a similar structure for modes. But why do we care the arrangement of modes? Why do we have two different modalities that "going-up" and "going-down", instead of one modality for "moving-from-one-to-the-other"? This is because these two modalities has different meanings.

Modalities for Distinction

For any two distinguished modes m_1 and m_2,

m_2
m_1

$\uparrow^{m_2}_{m_1}$ — A modality "going up" from m_1 to m_2

$\downarrow^{m_2}_{m_1}$ — A modality "going down" from m_2 to m_1

Modalities for Distinction
For any two distinguished modes m_1 and m_2,
m_2
m_1
$\uparrow^{m_2}_{m_1}$ — A modality "going up" from m_1 to m_2
$\downarrow^{m_2}_{m_1}$ — A modality "going down" from m_2 to m_1

Basically, when we have two distinguished modes, m_1 and m_2, we use two modalities to connect them: Going-up modality, and going-down modality. This can be applied to any of the previous modality examples. For metaprogramming, we can put code as an upper mode and program as a lower mode. For resource availability, a mode for unrestricted resources goes on top of the linear program. Resource privacy also has a similar structure for modes. But why do we care the arrangement of modes? Why do we have two different modalities that "going-up" and "going-down", instead of one modality for "moving-from-one-to-the-other"? This is because these two modalities has different meanings.

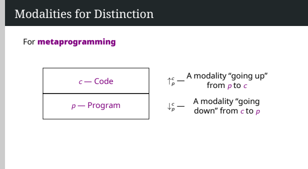
Modalities for Distinction

For **metaprogramming**

c — Code
p — Program

$\uparrow^{c}_{p}$ — A modality "going up" from p to c

$\downarrow^{c}_{p}$ — A modality "going down" from c to p



Basically, when we have two distinguished modes, m_1 and m_2, we use two modalities to connect them: Going-up modality, and going-down modality. This can be applied to any of the previous modality examples. For metaprogramming, we can put code as an upper mode and program as a lower mode. For resource availability, a mode for unrestricted resources goes on top of the linear program. Resource privacy also has a similar structure for modes. But why do we care the arrangement of modes? Why do we have two different modalities that "going-up" and "going-down", instead of one modality for "moving-from-one-to-the-other"? This is because these two modalities has different meanings.

Modalities for Distinction

For **resource availability**

u — Unrestricted Resource
p — Linear Program

$\uparrow^u_p$ — A modality "going up" from p to u

$\downarrow^u_p$ — A modality "going down" from u to p

Modalities for Distinction
For resource availability
u — Unrestricted Resource
p — Linear Program
$\uparrow^u_p$ — A modality "going up" from p to u
$\downarrow^u_p$ — A modality "going down" from u to p

Basically, when we have two distinguished modes, m_1 and m_2, we use two modalities to connect them: Going-up modality, and going-down modality. This can be applied to any of the previous modality examples. For metaprogramming, we can put code as an upper mode and program as a lower mode. For resource availability, a mode for unrestricted resources goes on top of the linear program. Resource privacy also has a similar structure for modes. But why do we care the arrangement of modes? Why do we have two different modalities that "going-up" and "going-down", instead of one modality for "moving-from-one-to-the-other"? This is because these two modalities has different meanings.

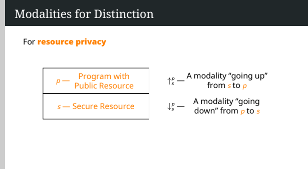
Modalities for Distinction

For **resource privacy**

p — Program with Public Resource
s — Secure Resource

$\uparrow^p_s$ — A modality "going up" from s to p

$\downarrow^p_s$ — A modality "going down" from p to s



Basically, when we have two distinguished modes, m_1 and m_2, we use two modalities to connect them: Going-up modality, and going-down modality. This can be applied to any of the previous modality examples. For metaprogramming, we can put code as an upper mode and program as a lower mode. For resource availability, a mode for unrestricted resources goes on top of the linear program. Resource privacy also has a similar structure for modes. But why do we care the arrangement of modes? Why do we have two different modalities that "going-up" and "going-down", instead of one modality for "moving-from-one-to-the-other"? This is because these two modalities has different meanings.

Going-up Modality

$\mathrm{lift}^{m_2}_{m_1}(M)$ — Lift the expression M from m_1 to its AST in m_2
Has the type $\uparrow^{m_2}_{m_1} A$ in m_2 if $M : A$ in m_1

Going-up Modality
$\mathrm{lift}^{m_2}_{m_1}(M)$ — Lift the expression M from m_1 to its AST in m_2
Has the type $\uparrow^{m_2}_{m_1} A$ in m_2 if $M : A$ in m_1

"Going-up" modality allows an upper mode to treat an expression from a lower mode as an AST. Using the syntax "lift", we lift the expression M from mode m_1 to its AST living in m_2. When we have this AST, we can either execute it in the original mode m_1 or splice it into another AST. Here, we use the syntax "unlift" for that.

Going-up Modality

$\mathrm{lift}_{m_1}^{m_2}(M)$ — Lift the expression M from m_1 to its AST in m_2
Has the type $\uparrow_{m_1}^{m_2} A$ in m_2 if $M : A$ in m_1

$\mathrm{unlift}_{m_1}^{m_2}(M)$ — Execute/splice-in the value (an AST) of M

Going-up Modality
$\mathrm{lift}_{m_1}^{m_2}(M)$ — Lift the expression M from m_1 to its AST in m_2
Has the type $\uparrow_{m_1}^{m_2} A$ in m_2 if $M : A$ in m_1
$\mathrm{unlift}_{m_1}^{m_2}(M)$ — Execute/splice-in the value (an AST) of M

"Going-up" modality allows an upper mode to treat an expression from a lower mode as an AST. Using the syntax "lift", we lift the expression M from mode m_1 to its AST living in m_2. When we have this AST, we can either execute it in the original mode m_1 or splice it into another AST. Here, we use the syntax "unlift" for that.

Going-down Modality

$\text{return}^{m_2}_{m_1}(M)$ — Store the expression M in m_2 for future use in m_2
Has the type $\downarrow^{m_2}_{m_1} A$ in m_1 if $M : A$ in m_2

$\text{let return}^{m_2}_{m_1} x = M \text{ in } N$ — Load the stored value pointed by M into x
and continue with N

Going-down Modality
$\text{return}^{m_2}_{m_1}(M)$ — Store the expression M in m_2 for future use in m_2
Has the type $\downarrow^{m_2}_{m_1} A$ in m_1 if $M : A$ in m_2
$\text{let return}^{m_2}_{m_1} x = M \text{ in } N$ — Load the stored value pointed by M into x
and continue with N

On the other hand, "Going-down" modality allows one to store an expression from an upper mode and use it later. Using the syntax "return", we store the value of expression M from mode m_2 and return a pointer of mode m_2 for the stored location. When we want to load it back, we use "let-return" on the pointer M, and it allows N to access the stored value. This pair of modalities allows us to construct other modalities using their combinations. Let's see a simple example for the case of metaprogramming.

Metaprogramming Example

```
pow : nat →  ↓ᶜₚ ↑ᶜₚ(nat → nat)
pow 0       = returnᶜₚ (liftᶜₚ (fun x → 1))
pow (suc n) =
  let returnᶜₚ C = pow n in
  returnᶜₚ (liftᶜₚ (fun x → x * (unliftᶜₚ (C)) x))
```

Modes

c — Code
p — Program

Metaprogramming Example

pow : nat → ↓ᶜₚ↑ᶜₚ(nat → nat)
pow 0 = returnᶜₚ (liftᶜₚ (fun x → 1))
pow (suc n) =
let returnᶜₚ C = pow n in
returnᶜₚ (liftᶜₚ (fun x → x * (unliftᶜₚ (C)) x))

Modes
c — Code
p — Program

I will use a code generator for the power function as an example. When we pass a number n to this function "pow", it returns a pointer to a code fragment for a function computing n-th power of the input argument. When the power is 0, we return a pointer to a code fragment, that, returns 1 for any argument. When the power is the successor of n, we first load a code fragment to compute n-th power, and then splice it into a code fragment with one multiplication of the parameter x. When we call this, for example, with the number 2, this generator gives a code fragment like this, which essentially computes x times x for the input x. This is the traditional implementation of the power function, but we have one issue: we need $O(n)$ number of stores and loads for every call of *pow*! Can we do better?

Metaprogramming Example

```
pow : nat → ↓^c_p ↑^c_p(nat → nat)
pow 0       = return^c_p (lift^c_p (fun x → 1))
pow (suc n) =
  let return^c_p C = pow n in
  return^c_p (lift^c_p (fun x → x * (unlift^c_p (C)) x))
```

Modes

c — Code
p — Program

Metaprogramming Example

I will use a code generator for the power function as an example. When we pass a number n to this function "pow", it returns a pointer to a code fragment for a function computing n-th power of the input argument. When the power is 0, we return a pointer to a code fragment, that, returns 1 for any argument. When the power is the successor of n, we first load a code fragment to compute n-th power, and then splice it into a code fragment with one multiplication of the parameter x. When we call this, for example, with the number 2, this generator gives a code fragment like this, which essentially computes x times x for the input x. This is the traditional implementation of the power function, but we have one issue: we need $O(n)$ number of stores and loads for every call of *pow*! Can we do better?

Metaprogramming Example

$$
\begin{array}{l}
\texttt{pow : nat} \to \downarrow^{c}_{p}\uparrow^{c}_{p}(\texttt{nat} \to \texttt{nat}) \\
\texttt{pow 0} \quad\quad\;\; = \texttt{return}^{c}_{p}\ (\texttt{lift}^{c}_{p}\ (\texttt{fun x} \to \texttt{1})) \\
\texttt{pow (suc n) =} \\
\quad \texttt{let return}^{c}_{p}\ \texttt{C = pow n in} \\
\quad \texttt{return}^{c}_{p}\ (\texttt{lift}^{c}_{p}\ (\texttt{fun x} \to \texttt{x * (unlift}^{c}_{p}\ \texttt{(C)) x))}
\end{array}
$$

Modes
c — Code
p — Program

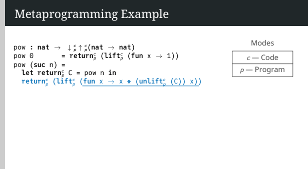
I will use a code generator for the power function as an example. When we pass a number n to this function "pow", it returns a pointer to a code fragment for a function computing n-th power of the input argument. When the power is 0, we return a pointer to a code fragment, that, returns 1 for any argument. When the power is the successor of n, we first load a code fragment to compute n-th power, and then splice it into a code fragment with one multiplication of the parameter x. When we call this, for example, with the number 2, this generator gives a code fragment like this, which essentially computes x times x for the input x. This is the traditional implementation of the power function, but we have one issue: we need $O(n)$ number of stores and loads for every call of *pow*! Can we do better?

Metaprogramming Example

```
pow : nat → ↓_p^c ↑_p^c(nat → nat)
pow 0       = return_p^c (lift_p^c (fun x → 1))
pow (suc n) =
  let return_p^c C = pow n in
  return_p^c (lift_p^c (fun x → x * (unlift_p^c (C)) x))
```

Modes

c — Code
p — Program



I will use a code generator for the power function as an example. When we pass a number n to this function "pow", it returns a pointer to a code fragment for a function computing n-th power of the input argument. When the power is 0, we return a pointer to a code fragment, that, returns 1 for any argument. When the power is the successor of n, we first load a code fragment to compute n-th power, and then splice it into a code fragment with one multiplication of the parameter x. When we call this, for example, with the number 2, this generator gives a code fragment like this, which essentially computes x times x for the input x. This is the traditional implementation of the power function, but we have one issue: we need $O(n)$ number of stores and loads for every call of *pow*! Can we do better?

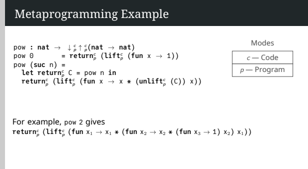
Metaprogramming Example

$$
\begin{array}{l}
\texttt{pow} : \textbf{nat} \to \ \downarrow^{c}_{p} \uparrow^{c}_{p}(\textbf{nat} \to \textbf{nat}) \\
\texttt{pow 0} \qquad\quad = \textbf{return}^{c}_{p}\ (\textbf{lift}^{c}_{p}\ (\textbf{fun}\ \texttt{x} \to \texttt{1})) \\
\texttt{pow}\ (\textbf{suc}\ \texttt{n}) = \\
\quad \textbf{let return}^{c}_{p}\ \texttt{C} = \texttt{pow n}\ \textbf{in} \\
\quad \textbf{return}^{c}_{p}\ (\textbf{lift}^{c}_{p}\ (\textbf{fun}\ \texttt{x} \to \texttt{x} * (\textbf{unlift}^{c}_{p}\ (\texttt{C}))\ \texttt{x}))
\end{array}
$$

Modes
c — Code
p — Program

For example, `pow 2` gives

$$
\textbf{return}^{c}_{p}\ (\textbf{lift}^{c}_{p}\ (\textbf{fun}\ \texttt{x}_1 \to \texttt{x}_1 * (\textbf{fun}\ \texttt{x}_2 \to \texttt{x}_2 * (\textbf{fun}\ \texttt{x}_3 \to \texttt{1})\ \texttt{x}_2)\ \texttt{x}_1))
$$



I will use a code generator for the power function as an example. When we pass a number n to this function "pow", it returns a pointer to a code fragment for a function computing n-th power of the input argument. When the power is 0, we return a pointer to a code fragment, that, returns 1 for any argument. When the power is the successor of n, we first load a code fragment to compute n-th power, and then splice it into a code fragment with one multiplication of the parameter x. When we call this, for example, with the number 2, this generator gives a code fragment like this, which essentially computes x times x for the input x. This is the traditional implementation of the power function, but we have one issue: we need $O(n)$ number of stores and loads for every call of *pow*! Can we do better?

Metaprogramming Example

$$
\begin{array}{l}
\texttt{pow : }\mathbf{nat} \to \downarrow^{c}_{p}\uparrow^{c}_{p}(\mathbf{nat} \to \mathbf{nat}) \\
\texttt{pow 0} \qquad\quad = \mathbf{return}^{c}_{p}\ (\mathbf{lift}^{c}_{p}\ (\mathbf{fun}\ \texttt{x} \to \texttt{1})) \\
\texttt{pow (}\mathbf{suc}\ \texttt{n) =} \\
\quad \mathbf{let}\ \mathbf{return}^{c}_{p}\ \texttt{C = pow n}\ \mathbf{in} \\
\quad \mathbf{return}^{c}_{p}\ (\mathbf{lift}^{c}_{p}\ (\mathbf{fun}\ \texttt{x} \to \texttt{x} * (\mathbf{unlift}^{c}_{p}\ \texttt{(C)) x))}
\end{array}
$$

Modes

c — Code
p — Program

For example, `pow 2` gives

$$\mathbf{return}^{c}_{p}\ (\mathbf{lift}^{c}_{p}\ (\mathbf{fun}\ \texttt{x}_1 \to \texttt{x}_1 * (\mathbf{fun}\ \texttt{x}_2 \to \texttt{x}_2 * (\mathbf{fun}\ \texttt{x}_3 \to \texttt{1})\ \texttt{x}_2)\ \texttt{x}_1))$$

Metaprogramming Example

I will use a code generator for the power function as an example. When we pass a number n to this function "pow", it returns a pointer to a code fragment for a function computing n-th power of the input argument. When the power is 0, we return a pointer to a code fragment, that, returns 1 for any argument. When the power is the successor of n, we first load a code fragment to compute n-th power, and then splice it into a code fragment with one multiplication of the parameter x. When we call this, for example, with the number 2, this generator gives a code fragment like this, which essentially computes x times x for the input x. This is the traditional implementation of the power function, but we have one issue: we need $O(n)$ number of stores and loads for every call of *pow*! Can we do better?

Metaprogramming Example

$\texttt{pow : nat} \to \downarrow^{c}_{p}\uparrow^{c}_{p}(\texttt{nat} \to \texttt{nat})$
$\texttt{pow 0} = \texttt{return}^{c}_{p}\ (\texttt{lift}^{c}_{p}\ (\texttt{fun x} \to \texttt{1}))$
$\texttt{pow (suc n)} =$
$\quad \texttt{let return}^{c}_{p}\ \texttt{C = pow n in}$
$\quad \texttt{return}^{c}_{p}\ (\texttt{lift}^{c}_{p}\ (\texttt{fun x} \to \texttt{x} * (\texttt{unlift}^{c}_{p}\ (\texttt{C}))\ \texttt{x}))$

Modes
c — Code
p — Program

For example, pow 2 gives
$\texttt{return}^{c}_{p}\ (\texttt{lift}^{c}_{p}\ (\texttt{fun}\ x_1 \to x_1 * (\texttt{fun}\ x_2 \to x_2 * (\texttt{fun}\ x_3 \to 1)\ x_2)\ x_1))$

But this requires $O(n)$ store-loads for every call!

Metaprogramming Example

I will use a code generator for the power function as an example. When we pass a number n to this function "pow", it returns a pointer to a code fragment for a function computing n-th power of the input argument. When the power is 0, we return a pointer to a code fragment, that, returns 1 for any argument. When the power is the successor of n, we first load a code fragment to compute n-th power, and then splice it into a code fragment with one multiplication of the parameter x. When we call this, for example, with the number 2, this generator gives a code fragment like this, which essentially computes x times x for the input x. This is the traditional implementation of the power function, but we have one issue: we need $O(n)$ number of stores and loads for every call of *pow*! Can we do better?

Separation of Concern in Metaprogramming

```
powHelper : nat → ↑ᶜₚ(nat → nat)
powHelper 0       = liftᶜₚ (fun x → 1)
powHelper (suc n) =
  liftᶜₚ (fun x → x * (unliftᶜₚ (powHelper n)) x))

pow : ↓ᶜₚnat → ↓ᶜₚ↑ᶜₚ(nat → nat)
pow dn =
  let returnᶜₚ n = dn in
  returnᶜₚ (powHelper n)
```

Modes
c — Code
p — Program

Separation of Concern in Metaprogramming

With explicit modes, yes, we can give a better implementation by separating concerns. Here, "powHelper" is a function to composing a code fragment for n-th power in the mode c. As we live in mode c during the composition, we can directly call "powHelper" in the code fragment without unnecessary store-load. When we provide the actual function "pow", we need only one load and one store. Here, we accept a pointer to nat living in mode c, so that we can use it in "powHelper". Thus, once we store a natural number and get a pointer to it, we no longer have $O(n)$ store-loads like the previous example. In other words, this system provides a more fine-grained control over when store-load happens, and how we process code fragments. Now, let's go into more details of this system.

Separation of Concern in Metaprogramming

$$
\begin{array}{l}
\mathtt{powHelper}\ :\ \mathbf{nat}\ \to\ \uparrow^{c}_{p}(\mathbf{nat}\ \to\ \mathbf{nat}) \\
\mathtt{powHelper}\ 0\qquad\quad =\ \mathbf{lift}^{c}_{p}\ (\mathbf{fun}\ \mathtt{x}\ \to\ 1) \\
\mathtt{powHelper}\ (\mathbf{suc}\ \mathtt{n})\ = \\
\quad \mathbf{lift}^{c}_{p}\ (\mathbf{fun}\ \mathtt{x}\ \to\ \mathtt{x}\ *\ (\mathbf{unlift}^{c}_{p}\ (\mathtt{powHelper}\ \mathtt{n}))\ \mathtt{x})) \\
\\
\mathtt{pow}\ :\ \downarrow^{c}_{p}\mathbf{nat}\ \to\ \downarrow^{c}_{p}\uparrow^{c}_{p}(\mathbf{nat}\ \to\ \mathbf{nat}) \\
\mathtt{pow}\ \mathtt{dn}\ = \\
\quad \mathbf{let}\ \mathbf{return}^{c}_{p}\ \mathtt{n}\ =\ \mathtt{dn}\ \mathbf{in} \\
\quad \mathbf{return}^{c}_{p}\ (\mathtt{powHelper}\ \mathtt{n})
\end{array}
$$

Direct recursive call and composition

Modes

c — Code
p — Program

Separation of Concern in Metaprogramming

With explicit modes, yes, we can give a better implementation by separating concerns. Here, "powHelper" is a function to composing a code fragment for n-th power in the mode c. As we live in mode c during the composition, we can directly call "powHelper" in the code fragment without unnecessary store-load. When we provide the actual function "pow", we need only one load and one store. Here, we accept a pointer to nat living in mode c, so that we can use it in "powHelper". Thus, once we store a natural number and get a pointer to it, we no longer have $O(n)$ store-loads like the previous example. In other words, this system provides a more fine-grained control over when store-load happens, and how we process code fragments. Now, let's go into more details of this system.

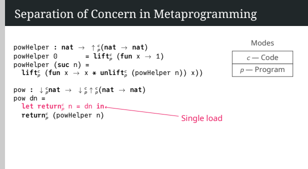
Separation of Concern in Metaprogramming

```
powHelper : nat → ↑^c_p(nat → nat)
powHelper 0       = lift^c_p (fun x → 1)
powHelper (suc n) =
  lift^c_p (fun x → x * unlift^c_p (powHelper n)) x))

pow : ↓^c_p nat → ↓^c_p ↑^c_p(nat → nat)
pow dn =
  let return^c_p n = dn in      ← Single load
  return^c_p (powHelper n)
```

Modes

c — Code
p — Program



With explicit modes, yes, we can give a better implementation by separating concerns. Here, "powHelper" is a function to composing a code fragment for n-th power in the mode c. As we live in mode c during the composition, we can directly call "powHelper" in the code fragment without unnecessary store-load. When we provide the actual function "pow", we need only one load and one store. Here, we accept a pointer to nat living in mode c, so that we can use it in "powHelper". Thus, once we store a natural number and get a pointer to it, we no longer have $O(n)$ store-loads like the previous example. In other words, this system provides a more fine-grained control over when store-load happens, and how we process code fragments. Now, let's go into more details of this system.

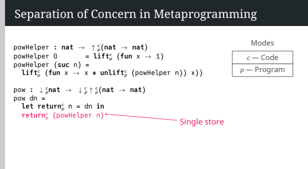
Separation of Concern in Metaprogramming

```
powHelper : nat → ↑^c_p(nat → nat)
powHelper 0       = lift^c_p (fun x → 1)
powHelper (suc n) =
  lift^c_p (fun x → x * unlift^c_p (powHelper n)) x))

pow : ↓^c_p nat → ↓^c_p ↑^c_p(nat → nat)
pow dn =
  let return^c_p n = dn in
  return^c_p (powHelper n)
```

Single store

Modes
c — Code
p — Program



With explicit modes, yes, we can give a better implementation by separating concerns. Here, “powHelper” is a function to composing a code fragment for n-th power in the mode c. As we live in mode c during the composition, we can directly call “powHelper” in the code fragment without unnecessary store-load. When we provide the actual function “pow”, we need only one load and one store. Here, we accept a pointer to nat living in mode c, so that we can use it in “powHelper”. Thus, once we store a natural number and get a pointer to it, we no longer have $O(n)$ store-loads like the previous example. In other words, this system provides a more fine-grained control over when store-load happens, and how we process code fragments. Now, let’s go into more details of this system.

Separation of Concern in Metaprogramming

```
powHelper : nat → ↑^c_p(nat → nat)
powHelper 0       = lift^c_p (fun x → 1)
powHelper (suc n) =
  lift^c_p (fun x → x * unlift^c_p (powHelper n)) x))

pow : ↓^c_p nat → ↓^c_p ↑^c_p(nat → nat)
pow dn =
  let return^c_p n = dn in
  return^c_p (powHelper n)
```

Modes
c — Code
p — Program

Separate `nat` store/loads from the composition of code fragments

Separation of Concern in Metaprogramming
Separate nat store/loads from the composition of code fragments

With explicit modes, yes, we can give a better implementation by separating concerns. Here, "powHelper" is a function to composing a code fragment for n-th power in the mode c. As we live in mode c during the composition, we can directly call "powHelper" in the code fragment without unnecessary store-load. When we provide the actual function "pow", we need only one load and one store. Here, we accept a pointer to nat living in mode c, so that we can use it in "powHelper". Thus, once we store a natural number and get a pointer to it, we no longer have $O(n)$ store-loads like the previous example. In other words, this system provides a more fine-grained control over when store-load happens, and how we process code fragments. Now, let's go into more details of this system.

Syntax

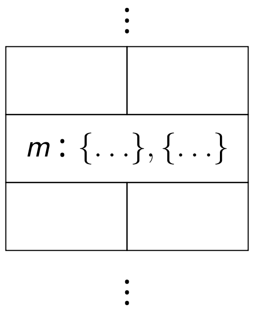
Modes	m, n, l, h		
Types	S, T	$:=$	$\uparrow^n_l S \mid \downarrow^h_n S \mid S \multimap T \mid \ldots$
Terms	L, M	$:=$	$x \mid \mathsf{lift}^n_l\ (L) \mid \mathsf{unlift}^n_l\ (L)$
		$\mid$	$\mathsf{return}^h_n\ (L) \mid \mathsf{let\ return}^h_n\ (x) = L \mathsf{\ in\ } M$
		$\mid$	$\lambda(x{:}^n T).L \mid L\ M \mid \ldots$
Context	Γ	$:=$	$\cdot \mid \Gamma, x{:}^n S$

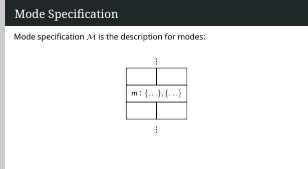
Syntax
Modes m, n, l, h
Types S, T $:=$ $\uparrow^n_l S \mid \downarrow^h_n S \mid S \multimap T \mid \ldots$
Terms L, M $:=$ $x \mid \mathsf{lift}^n_l\ (L) \mid \mathsf{unlift}^n_l\ (L)$
$\mid$ $\mathsf{return}^h_n\ (L) \mid \mathsf{let\ return}^h_n\ (x) = L \mathsf{\ in\ } M$
$\mid$ $\lambda(x{:}^n T).L \mid L\ M \mid \ldots$
Context Γ $:=$ $\cdot \mid \Gamma, x{:}^n S$

The syntax for this system is simple; we have going-up and going-down modalities indexed by departing and arriving modes, and possibly linear function type, as we want to capture resource availability as well, and introduction and elimination forms for these types. Note that we annotate each assumption with a mode. This mode will be used to validate that an upper mode expression depends on no lower mode assumptions, so that an upper mode expression can be stored independently of a lower mode environment. However, this syntax does not allow us to capture different varieties of modalities. The power comes from a parameter for this system, called mode specification.

Mode Specification

Mode specification $\mathcal{M}$ is the description for modes:

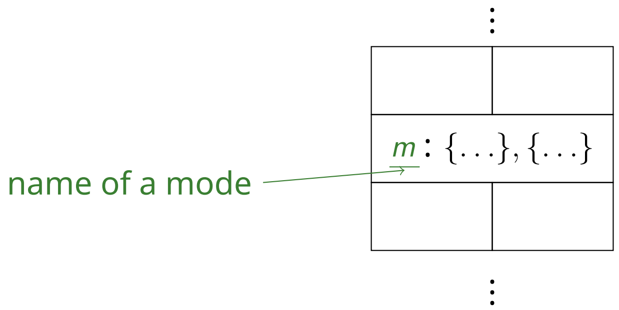


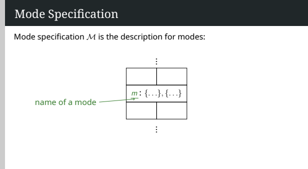


A mode specification curly M is the description for modes. In this description, we have, first, which modes are there, i.e. the names of modes. Second, which structural rules are allowed in each mode. Each mode can allow any combinations of contraction and weakening rules. We will use $\mathtt{st}_{\mathcal{M}}(m)$ to refer to this set of rules for mode m. Third, which types are allowed in each mode. We will use $\mathtt{op}_{\mathcal{M}}(m)$ to refer to this set of allowed types for mode m. Finally, we give which mode goes on top of other modes. This arrangement specifies when an expression in a mode can depend on an assumption of another mode, as well as when going-up and going-down modalities are possible. We refer to this ordering of modes using this less-than relation decorated with the mode specification. Let's see some example mode specifications.

Mode Specification

Mode specification $\mathcal{M}$ is the description for modes:

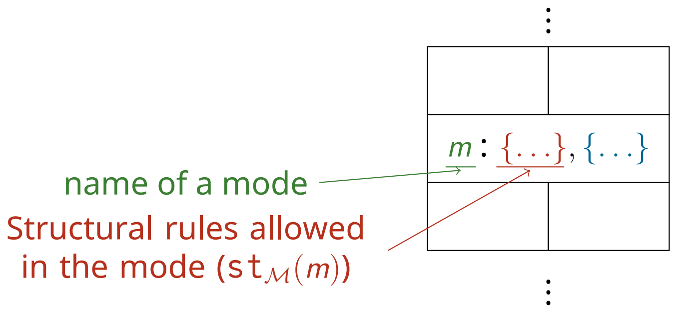


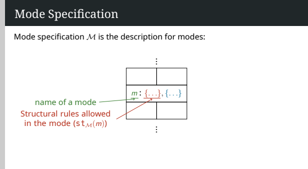


A mode specification curly M is the description for modes. In this description, we have, first, which modes are there, i.e. the names of modes. Second, which structural rules are allowed in each mode. Each mode can allow any combinations of contraction and weakening rules. We will use $\mathtt{st}_{\mathcal{M}}(m)$ to refer to this set of rules for mode m. Third, which types are allowed in each mode. We will use $\mathtt{op}_{\mathcal{M}}(m)$ to refer to this set of allowed types for mode m. Finally, we give which mode goes on top of other modes. This arrangement specifies when an expression in a mode can depend on an assumption of another mode, as well as when going-up and going-down modalities are possible. We refer to this ordering of modes using this less-than relation decorated with the mode specification. Let's see some example mode specifications.

Mode Specification

Mode specification $\mathcal{M}$ is the description for modes:

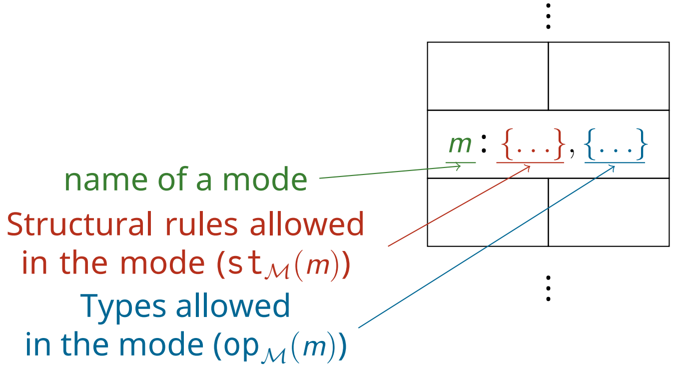


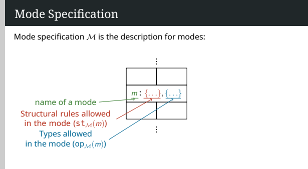


A mode specification curly M is the description for modes. In this description, we have, first, which modes are there, i.e. the names of modes. Second, which structural rules are allowed in each mode. Each mode can allow any combinations of contraction and weakening rules. We will use $\mathtt{st}_{\mathcal{M}}(m)$ to refer to this set of rules for mode m. Third, which types are allowed in each mode. We will use $\mathtt{op}_{\mathcal{M}}(m)$ to refer to this set of allowed types for mode m. Finally, we give which mode goes on top of other modes. This arrangement specifies when an expression in a mode can depend on an assumption of another mode, as well as when going-up and going-down modalities are possible. We refer to this ordering of modes using this less-than relation decorated with the mode specification. Let's see some example mode specifications.

Mode Specification

Mode specification $\mathcal{M}$ is the description for modes:

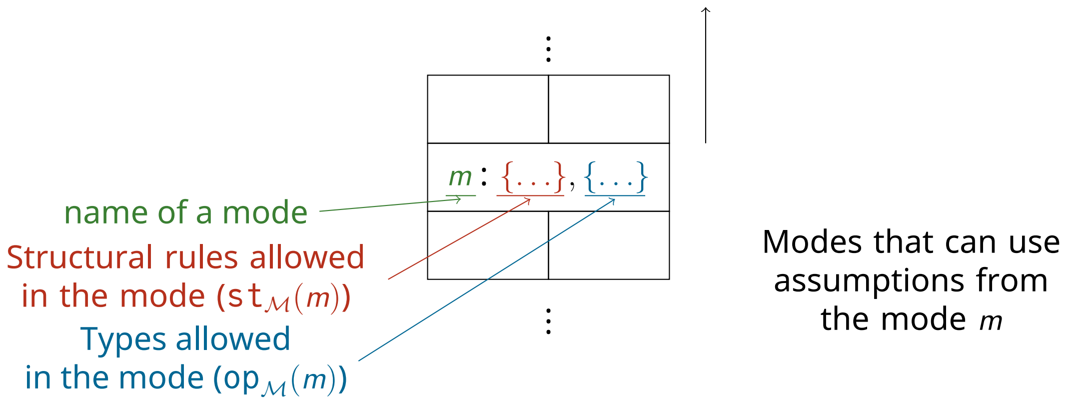


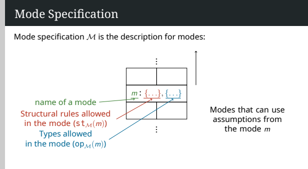


A mode specification curly M is the description for modes. In this description, we have, first, which modes are there, i.e. the names of modes. Second, which structural rules are allowed in each mode. Each mode can allow any combinations of contraction and weakening rules. We will use $\mathtt{st}_{\mathcal{M}}(m)$ to refer to this set of rules for mode m. Third, which types are allowed in each mode. We will use $\mathtt{op}_{\mathcal{M}}(m)$ to refer to this set of allowed types for mode m. Finally, we give which mode goes on top of other modes. This arrangement specifies when an expression in a mode can depend on an assumption of another mode, as well as when going-up and going-down modalities are possible. We refer to this ordering of modes using this less-than relation decorated with the mode specification. Let's see some example mode specifications.

Mode Specification

Mode specification $\mathcal{M}$ is the description for modes:

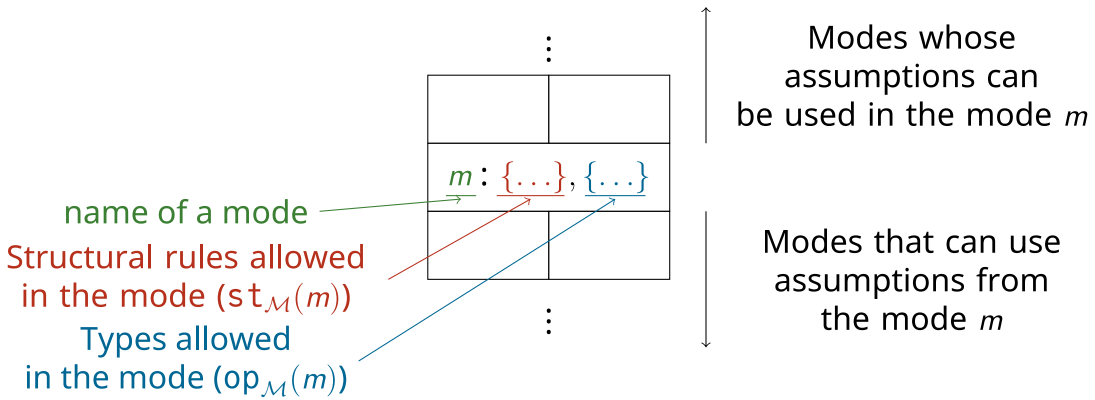


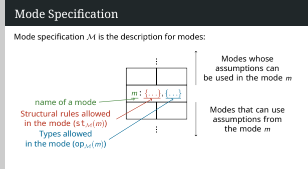


A mode specification curly M is the description for modes. In this description, we have, first, which modes are there, i.e. the names of modes. Second, which structural rules are allowed in each mode. Each mode can allow any combinations of contraction and weakening rules. We will use $\mathtt{st}_{\mathcal{M}}(m)$ to refer to this set of rules for mode m. Third, which types are allowed in each mode. We will use $\mathtt{op}_{\mathcal{M}}(m)$ to refer to this set of allowed types for mode m. Finally, we give which mode goes on top of other modes. This arrangement specifies when an expression in a mode can depend on an assumption of another mode, as well as when going-up and going-down modalities are possible. We refer to this ordering of modes using this less-than relation decorated with the mode specification. Let's see some example mode specifications.

Mode Specification

Mode specification $\mathcal{M}$ is the description for modes:

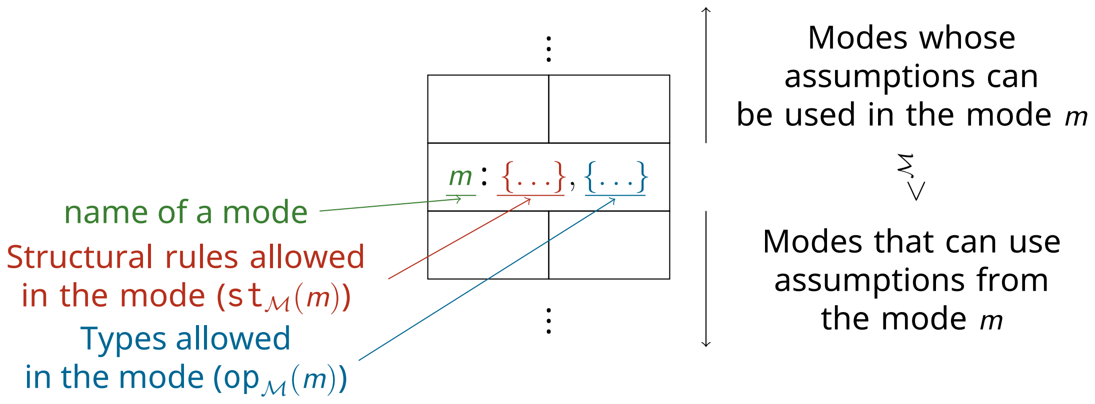


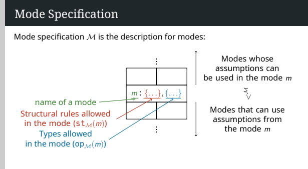


A mode specification curly M is the description for modes. In this description, we have, first, which modes are there, i.e. the names of modes. Second, which structural rules are allowed in each mode. Each mode can allow any combinations of contraction and weakening rules. We will use $\mathtt{st}_{\mathcal{M}}(m)$ to refer to this set of rules for mode m. Third, which types are allowed in each mode. We will use $\mathtt{op}_{\mathcal{M}}(m)$ to refer to this set of allowed types for mode m. Finally, we give which mode goes on top of other modes. This arrangement specifies when an expression in a mode can depend on an assumption of another mode, as well as when going-up and going-down modalities are possible. We refer to this ordering of modes using this less-than relation decorated with the mode specification. Let's see some example mode specifications.

Mode Specification

Mode specification $\mathcal{M}$ is the description for modes:





A mode specification curly M is the description for modes. In this description, we have, first, which modes are there, i.e. the names of modes. Second, which structural rules are allowed in each mode. Each mode can allow any combinations of contraction and weakening rules. We will use $\mathtt{st}_{\mathcal{M}}(m)$ to refer to this set of rules for mode m. Third, which types are allowed in each mode. We will use $\mathtt{op}_{\mathcal{M}}(m)$ to refer to this set of allowed types for mode m. Finally, we give which mode goes on top of other modes. This arrangement specifies when an expression in a mode can depend on an assumption of another mode, as well as when going-up and going-down modalities are possible. We refer to this ordering of modes using this less-than relation decorated with the mode specification. Let's see some example mode specifications.

Examples of Mode Specifications

$p : \{\textbf{Co}, \textbf{Wk}\}, \{\multimap\}$

STLC

$c : \{\textbf{Co}, \textbf{Wk}\}, \{\uparrow, \multimap\}$
$p : \{\textbf{Co}, \textbf{Wk}\}, \{\downarrow, \multimap\}$

Metaprogramming system
with the separation of concerns

$u : \{\textbf{Co}, \textbf{Wk}\}, \{\uparrow\}$
$p : \{\}, \{\downarrow, \multimap\}$

Linear type system with !

$p : \{\textbf{Co}, \textbf{Wk}\}, \{\uparrow, \multimap\}$
$s : \{\textbf{Co}, \textbf{Wk}\}, \{\downarrow\}$

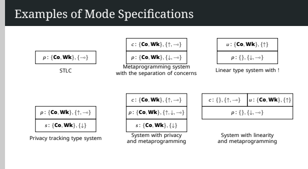
Privacy tracking type system

$c : \{\textbf{Co}, \textbf{Wk}\}, \{\uparrow, \multimap\}$
$p : \{\textbf{Co}, \textbf{Wk}\}, \{\uparrow, \downarrow, \multimap\}$
$s : \{\textbf{Co}, \textbf{Wk}\}, \{\downarrow\}$

System with privacy
and metaprogramming

$c : \{\}, \{\uparrow, \multimap\}$	$u : \{\textbf{Co}, \textbf{Wk}\}, \{\uparrow\}$
$p : \{\}, \{\downarrow, \multimap\}$	

System with linearity
and metaprogramming



We have the simplest mode specification for the first: one for Simply-typed lambda calculus. Here, we have only one mode, which allows no going-up or down modalities but function space, and all structural rules. If we allow two modes, c and p, where both have the function spaces, and each has the corresponding going-up or down modality. This gives a metaprogramming system allowing the previous "powHelper" example, where we distinguish two modes explicitly. By removing structural rules from a mode, we get a linear type system, and by putting a mode under the program mode p, we get a resource privacy. Furthermore, we can combine these modes in any ways.

Well-modedness Rules of Modalities

$$\boxed{\vdash^{n}_{\mathcal{M}} S : *}$$

$$\frac{\vdash^{l}_{\mathcal{M}} S : * \qquad l <^{\mathcal{M}} n \qquad \uparrow \in \mathsf{op}_{\mathcal{M}}(n)}{\vdash^{n}_{\mathcal{M}} \uparrow^{n}_{l} S : *}\ \mathtt{WM}\uparrow$$

$$\frac{\vdash^{h}_{\mathcal{M}} S : * \qquad n <^{\mathcal{M}} h \qquad \downarrow \in \mathsf{op}_{\mathcal{M}}(n)}{\vdash^{n}_{\mathcal{M}} \downarrow^{h}_{n} S : *}\ \mathtt{WM}\downarrow$$

Well-modedness Rules of Modalities

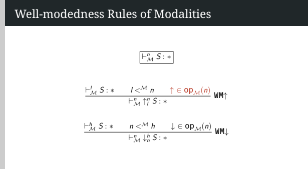
Once we have the mode specification, we can first check whether a type is valid in a mode. We call this well-modedness, and its judgement defined by these rules. Going-up modality is valid only when the modified type is valid in the departing mode and the destination mode allows the modality. Likewise for going-down modality.

Well-modedness Rules of Modalities

$$\boxed{\vdash^n_{\mathcal{M}} S : *}$$

$$\frac{\vdash^l_{\mathcal{M}} S : * \qquad l <^{\mathcal{M}} n \qquad {\color{red}\uparrow \in \mathsf{op}_{\mathcal{M}}(n)}}{\vdash^n_{\mathcal{M}} \uparrow^n_l S : *}\ \texttt{WM}\uparrow$$

$$\frac{\vdash^h_{\mathcal{M}} S : * \qquad n <^{\mathcal{M}} h \qquad \downarrow \in \mathsf{op}_{\mathcal{M}}(n)}{\vdash^n_{\mathcal{M}} \downarrow^h_n S : *}\ \texttt{WM}\downarrow$$



Once we have the mode specification, we can first check whether a type is valid in a mode. We call this well-modedness, and its judgement defined by these rules. Going-up modality is valid only when the modified type is valid in the departing mode and the destination mode allows the modality. Likewise for going-down modality.

Typing Rules of Modalities

$$\boxed{\Gamma \vdash^{n}_{\mathcal{M}} L : S}$$

$$\frac{\Gamma \vdash^{l}_{\mathcal{M}} L : S}{\Gamma \vdash^{n}_{\mathcal{M}} \mathsf{lift}^{n}_{l}\ (L) : \uparrow^{n}_{l} S}\ \mathrm{I}{\uparrow} \qquad \frac{\vdash^{h}_{\mathcal{M}} \Gamma \qquad \Gamma \vdash^{h}_{\mathcal{M}} L : \uparrow^{h}_{n} S}{\Gamma \vdash^{n}_{\mathcal{M}} \mathsf{unlift}^{h}_{n}\ (L) : S}\ \mathrm{E}{\uparrow}$$

$$\frac{\vdash^{h}_{\mathcal{M}} \Gamma \qquad \Gamma \vdash^{h}_{\mathcal{M}} L : S}{\Gamma \vdash^{n}_{\mathcal{M}} \mathsf{return}^{h}_{n}\ (L) : \downarrow^{h}_{n} S}\ \mathrm{I}{\downarrow} \qquad \frac{\Gamma' \vdash^{n}_{\mathcal{M}} L : \downarrow^{h}_{n} T \qquad \Gamma, x{:}^{h} T \vdash^{n}_{\mathcal{M}} M : S}{\Gamma, \Gamma' \vdash^{n}_{\mathcal{M}} \mathsf{let\ return}^{h}_{n}\ (x) = L \ \mathsf{in}\ M : S}\ \mathrm{E}{\downarrow}$$

Once we have well-moded types, we can check a term is well-typed under a context. Here, these premises check whether the context contains any invalid assumptions, i.e. assumptions from lower modes. Note that, the rule for let-return, which has two sub-expressions, splits context into two; this is to declaratively allow substructural modes such as linear modes.

Typing Rules of Modalities

$$\boxed{\Gamma \vdash^{n}_{\mathcal{M}} L : S}$$

$$\frac{\Gamma \vdash^{l}_{\mathcal{M}} L : S}{\Gamma \vdash^{n}_{\mathcal{M}} \mathsf{lift}^{n}_{l}\ (L) : \uparrow^{n}_{l} S}\ \mathrm{I}{\uparrow} \qquad \frac{\color{red}{\vdash^{h}_{\mathcal{M}} \Gamma} \qquad \Gamma \vdash^{h}_{\mathcal{M}} L : \uparrow^{h}_{n} S}{\Gamma \vdash^{n}_{\mathcal{M}} \mathsf{unlift}^{h}_{n}\ (L) : S}\ \mathrm{E}{\uparrow}$$

$$\frac{\color{red}{\vdash^{h}_{\mathcal{M}} \Gamma} \qquad \Gamma \vdash^{h}_{\mathcal{M}} L : S}{\Gamma \vdash^{n}_{\mathcal{M}} \mathsf{return}^{h}_{n}\ (L) : \downarrow^{h}_{n} S}\ \mathrm{I}{\downarrow} \qquad \frac{\Gamma' \vdash^{n}_{\mathcal{M}} L : \downarrow^{h}_{n} T \qquad \Gamma, x{:}^{h} T \vdash^{n}_{\mathcal{M}} M : S}{\Gamma, \Gamma' \vdash^{n}_{\mathcal{M}} \mathsf{let}\ \mathsf{return}^{h}_{n}\ (x) = L\ \mathsf{in}\ M : S}\ \mathrm{E}{\downarrow}$$

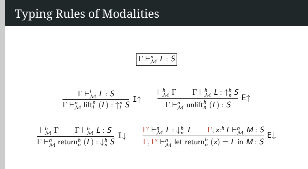
Once we have well-moded types, we can check a term is well-typed under a context. Here, these premises check whether the context contains any invalid assumptions, i.e. assumptions from lower modes. Note that, the rule for let-return, which has two sub-expressions, splits context into two; this is to declaratively allow substructural modes such as linear modes.

Typing Rules of Modalities

$$\boxed{\Gamma \vdash^n_{\mathcal{M}} L : S}$$

$$\frac{\Gamma \vdash^l_{\mathcal{M}} L : S}{\Gamma \vdash^n_{\mathcal{M}} \mathsf{lift}^n_l\ (L) : \uparrow^n_l S}\ \mathrm{I}{\uparrow} \qquad \frac{\vdash^h_{\mathcal{M}} \Gamma \qquad \Gamma \vdash^h_{\mathcal{M}} L : \uparrow^h_n S}{\Gamma \vdash^n_{\mathcal{M}} \mathsf{unlift}^h_n\ (L) : S}\ \mathrm{E}{\uparrow}$$

$$\frac{\vdash^h_{\mathcal{M}} \Gamma \qquad \Gamma \vdash^h_{\mathcal{M}} L : S}{\Gamma \vdash^n_{\mathcal{M}} \mathsf{return}^h_n\ (L) : \downarrow^h_n S}\ \mathrm{I}{\downarrow} \qquad \frac{\Gamma' \vdash^n_{\mathcal{M}} L : \downarrow^h_n T \qquad \Gamma, x{:}^h T \vdash^n_{\mathcal{M}} M : S}{\Gamma, \Gamma' \vdash^n_{\mathcal{M}} \mathsf{let\ return}^h_n\ (x) = L\ \mathsf{in}\ M : S}\ \mathrm{E}{\downarrow}$$



Once we have well-moded types, we can check a term is well-typed under a context. Here, these premises check whether the context contains any invalid assumptions, i.e. assumptions from lower modes. Note that, the rule for let-return, which has two sub-expressions, splits context into two; this is to declaratively allow substructural modes such as linear modes.

Typing Rules for Assumptions

$$\frac{}{\cdot, x{:}^n S \vdash^n_{\mathcal{M}} x : S}\ \mathtt{var}$$

$$\frac{\Gamma \vdash^n_{\mathcal{M}} L : S \qquad \mathtt{Wk} \in \mathtt{st}_{\mathcal{M}}(m)}{\Gamma, x{:}^m T \vdash^n_{\mathcal{M}} L : S}\ \mathtt{weaken}$$

$$\frac{\Gamma, x{:}^m T, x{:}^m T \vdash^n_{\mathcal{M}} L : S \qquad \mathtt{Co} \in \mathtt{st}_{\mathcal{M}}(m)}{\Gamma, x{:}^m T \vdash^n_{\mathcal{M}} L : S}\ \mathtt{contract}$$

Typing Rules for Assumptions

To deal with substructural modes, we have a variable rule that allows no other assumptions, and two structural rules allowed only in specified modes. These rules allow maximal freedom to this system, in terms of both resource availability and privacy.

Typing Rules for Assumptions

$$\frac{}{\cdot, x{:}^n S \vdash^n_{\mathcal{M}} x : S}\ \texttt{var}$$

$$\frac{\Gamma \vdash^n_{\mathcal{M}} L : S \qquad \texttt{Wk} \in \texttt{st}_{\mathcal{M}}(m)}{\Gamma, x{:}^m T \vdash^n_{\mathcal{M}} L : S}\ \texttt{weaken}$$

$$\frac{\Gamma, x{:}^m T, x{:}^m T \vdash^n_{\mathcal{M}} L : S \qquad \texttt{Co} \in \texttt{st}_{\mathcal{M}}(m)}{\Gamma, x{:}^m T \vdash^n_{\mathcal{M}} L : S}\ \texttt{contract}$$

Typing Rules for Assumptions

$$\frac{}{\cdot, x{:}^n S \vdash^n_{\mathcal{M}} x : S}\ \texttt{var}$$

$$\frac{\Gamma \vdash^n_{\mathcal{M}} L : S \qquad \texttt{Wk} \in \texttt{st}_{\mathcal{M}}(m)}{\Gamma, x{:}^m T \vdash^n_{\mathcal{M}} L : S}\ \texttt{weaken}$$

$$\frac{\Gamma, x{:}^m T, x{:}^m T \vdash^n_{\mathcal{M}} L : S \qquad \texttt{Co} \in \texttt{st}_{\mathcal{M}}(m)}{\Gamma, x{:}^m T \vdash^n_{\mathcal{M}} L : S}\ \texttt{contract}$$

To deal with substructural modes, we have a variable rule that allows no other assumptions, and two structural rules allowed only in specified modes. These rules allow maximal freedom to this system, in terms of both resource availability and privacy.

Other Works on Multi-modal Systems

- System for Session Types [Pruiksma and Pfenning 2021]
- Systems without metaprogramming support [Orchard et al. 2019,Choudhury et al. 2021,Moon et al. 2021]
- Systems without resource guarantees [Gratzer et al. 2020]
- Other System [Abel and Bernardy 2020]

Other Works on Multi-modal Systems
- System for Session Types [Pruiksma and Pfenning 2021]
- Systems without metaprogramming support [Orchard et al. 2019,Choudhury et al. 2021,Moon et al. 2021]
- Systems without resource guarantees [Gratzer et al. 2020]
- Other System [Abel and Bernardy 2020]

There are other works has been done on multi-modal systems. Pruiksma and Pfenning develope a system for session types using this two modality approach, but do not apply it to functional programming. Orchard et al, Choudhury et al, and Moon et al describe a systems for resource availability and privacy, but they come without metaprogramming support. Another multi-modal system from Gratzer et al comes without resource guarantees such as linearity. Abel and Bernardy's system is most comparable to this system in terms of its power, but their system has only one indexed modality instead of two, and thus they cannot implement, for example, "powHelper"-like efficient code generator.

Current Status of Project

- Preservation/progress are proved for declarative typing rules
- Implementation based on algorithmic typing rules
- Full embedding of $\lambda^{\Box}$ is proved and mechanized

Current Status of Project

- Preservation/progress are proved for declarative typing rules
- Implementation based on algorithmic typing rules
- Full embedding of $\lambda^{\Box}$ is proved and mechanized

For this system, we currently have preservation and progress proofs for declarative typing rules I introduced here, and an interpreter implementation based on algorithmic typing rules. Also, we proved that we can fully embed $\lambda^{\Box}$, one of the most commonly used bases of staged programming and metaprogramming, into this system. Thank you for listening, and please feel free to ask any questions!

References

- Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. *Proc. ACM Program. Lang.* 4, ICFP, Article 90 (aug 2020), 28 pages. `https://doi.org/10.1145/3408972`
- Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A Graded Dependent Type System with a Usage-Aware Semantics. *Proc. ACM Program. Lang.* 5, POPL, Article 50 (jan 2021), 32 pages. `https://doi.org/10.1145/3434331`
- Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (Saarbrücken, Germany) (LICS '20)*. Association for Computing Machinery, New York, NY, USA, 492–506. `https://doi.org/10.1145/3373718.3394736`
- Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. In *Programming Languages and Systems*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 462–490.
- Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (jul 2019), 30 pages. `https://doi.org/10.1145/3341714`
- Klaas Pruiksma and Frank Pfenning. 2021. A message-passing interpretation of adjoint logic. *Journal of Logical and Algebraic Methods in Programming* 120 (4 2021), 100637. `https://doi.org/10.1016/J.JLAMP.2020.100637`