

# Incremental Type-Checking for Metaprograms

Weiyu Miao

# Introduction to Metaprogramming

- Metaprogramming is a programming technique for generating and manipulating other programs.
- MetaML, MetaOCaml, Haskell Templates, C++ Templates, etc
- Code generation using C++ templates.

```
template <int N>  
int powN(int M) { return M * powN<N-1>(M); }
```

```
template<>  
int powN<0>(int M) { return 1; }
```

```
template int powN<2>(int);
```

**at compile time**, it generates the code  $M * M * 1$

- Meta Evaluation

# Introduction to Metaprogramming (Cont.)

Reflective metaprogramming<sup>1</sup> language syntax:

- The language in which the metaprogram is written is called the **meta** language.

meta language

$e^m ::= x \mid c^m \mid \lambda x : \tau^m . e^m \mid e^m \ e^m \mid \langle e^c \rangle \mid \%e^m \mid \gamma \mid e^m \rightarrow e^m \mid$   
 $\rightarrow^? \mid \mathbf{dom} \ e^m \mid \mathbf{cod} \ e^m \mid \mathbf{typeof} \ e^m \mid ==_{\tau} \mid \dots$

meta type

$\tau^m ::= \mathbf{type} \mid \gamma \mid \mathbf{code} \ \gamma \mid \tau^m \rightarrow \tau^m$

- The language of the programs that are manipulated is called the **object** language.

object language

$e^c ::= x \mid c^c \mid \lambda x : e^m . e^c \mid e^c \ e^c \mid \sim e^m \mid \dots$

---

<sup>1</sup>supported by NSF, CCF-0702362

- Partial evaluation.

```
template <typename T, int U>  
T powerN (T m)  
{ return m * powerN<T, U-1>(m) }
```

```
let rec meta powerN =  
  λ T : type. λ U : int.  
    <λ m : T.  
      m * ((~ powerN T (U-1)) m)>
```

- Type reflection.

```
let meta rec type2string =  
  λ t : type.  
    if t ==τ int then "int" else  
      if t ==τ bool then "bool" else  
        if →? t then  
          let meta t1 = type2string (dom t) in  
          let meta t2 = type2string (cod t) in  
            t1 + "->" + t2  
in  
  let f = λ x : int. λ y : bool. x in  
    (type2string (typeof <f>))
```

- Dependent types.

```
let meta fun =  
  λ X : type. λ Y : bool. λ Z : type.  
    <λ a : X.  
      λ b : if Y then (int → int) else int.  
        λ c : Z.  
          if a then (c b) else b>  
in  
  (fun bool false (int → int))
```

# System F Style

```
let meta fun =  
  λ X : type. λ Y : bool. λ Z : type.  
    <λ a : X.  
      λ b : if Y then (int → int) else int.  
        λ c : Z.  
          if a then (c b) else b>  
  
(fun bool true ((int → int) → int → int))  
  
(fun bool false (int → bool))
```

☞ Type-checking at definition time – before meta evaluation.

Suppose  $\Gamma = X : \text{type}, Y : \text{bool}, Z : \text{type}$  and  $\Delta = a : X, b :$   
 $\text{if } Y \text{ then } (\text{int} \rightarrow \text{int}) \text{ else int}, c : Z$

(1)  $\Gamma, \Delta \not\vdash a : \text{bool}$

(2)  $\Gamma, \Delta \not\vdash c : (\text{typeof } b) \rightarrow (\text{typeof } b)$

## System F Style (Cont.)

drawback : precluding well-typed expressions.

```
let meta fun =  
  λ X : type. λ Y : bool. λ Z : type.  
    <λ a : X.  
      λ b : if Y then (int → int) else int.  
        λ c : Z.  
          if a then (c b) else b>  
  
(fun bool false (int → int))  
  
(fun bool true ((int → int) → int → int))
```

# C++ Templates Style

```
let meta fun =  
  λ X : type. λ Y : bool. λ Z : type.  
    <λ a : X.  
      λ b : if Y then (int → int) else int.  
        λ c : Z.  
          if a then (c b) else b>  
  
(fun bool true ((int → int) → int → int)) ...  
  
(fun bool false (int → bool)) ...
```

☞ Type-checking at instantiation time – after meta evaluation<sup>2</sup>.

✓ (1)  $\langle \lambda a : \text{bool}. \lambda b : \text{int} \rightarrow \text{int}. \lambda c : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}. \text{if } a \text{ then } (c \ b) \text{ else } b \rangle$

✗ (2)  $\langle \lambda a : \text{bool}. \lambda b : \text{int}. \lambda c : \text{int} \rightarrow \text{bool}. \text{if } a \text{ then } (c \ b) \text{ else } b \rangle$

<sup>2</sup>Ronald Garcia. Static Computation and Reflection. PhD thesis, Indiana University, September 2008.

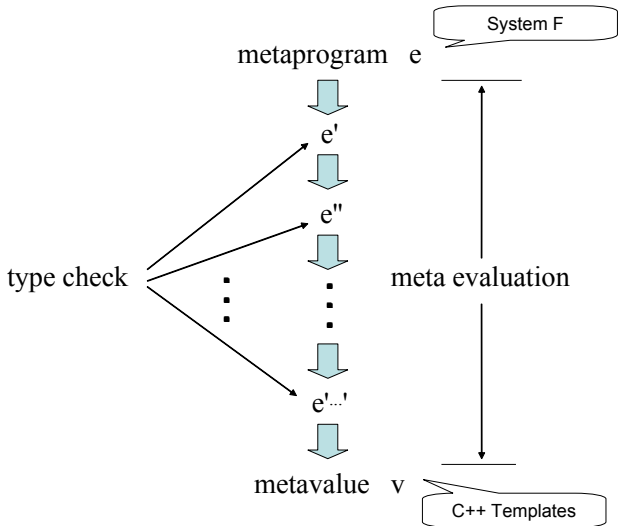
# C++ Templates Style (Cont.)

drawback : inefficiency.

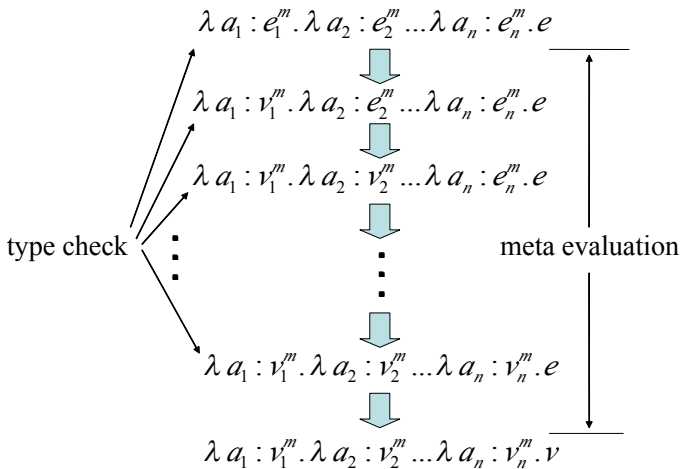
```
let meta fun =  
  λ X : bool.  
    <λ a0 : if X then int else bool.  
      λ a1 : e1m. λ a2 : e2m. ⋯ λ an : enm.  
        let x = em in /* Θ(em) = 2n */  
          if a0 then ⋯ else ⋯>
```

```
fun true =  
  <λ a0 : int.  
    λ a1 : v1m. a2 : v2m. ⋯ an : vnm.  
      let x = vm in  
        if a0 then ⋯ else ⋯>
```

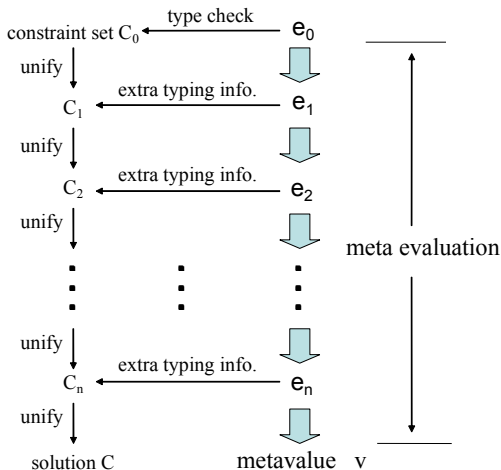
# Idea : Incremental Type-Checking



# Idea : Incremental Type-Checking (Cont.)



# Incremental Type-Checking Based on Unification



# Example

```
let meta fun =  
  λ X : type. λ Y : bool. λ Z : type.  
    <λ a : (X)α.  
      λ b : (if Y then int → int else int)β.  
        λ c : (Z)γ.  
          if a then (c b) else b>
```

Suppose  $\Gamma = X : \text{type}, Y : \text{bool}, Z : \text{type}$  and  $\Delta = a : \alpha, b : \beta, c : \gamma$

$$\frac{\Gamma, \Delta \vdash c : \gamma \mid \{\} \quad \Gamma, \Delta \vdash b : \beta \mid \{\} \quad a' \text{ is fresh}}{\Gamma, \Delta \vdash (c \ b) : a' \mid \{\gamma = \beta \rightarrow a'\}}$$
$$\frac{\Gamma, \Delta \vdash a : \alpha \quad \Gamma, \Delta \vdash (c \ b) : a' \mid \{\gamma = \beta \rightarrow a'\} \quad \Gamma, \Delta \vdash b : \beta \mid \{\}}{\Gamma, \Delta \vdash (\text{if } a \text{ then } (c \ b) \text{ else } b) : a' \mid \{\alpha = \text{bool}, \gamma = \beta \rightarrow a', a' = \beta\}}$$
$$\text{unify } (\{\alpha = \text{bool}, \gamma = \beta \rightarrow a', a' = \beta\}) = \{\alpha = \text{bool}, \gamma = \beta \rightarrow \beta, a' = \beta\}$$

## Example (Cont.)

```
let meta fun =  
  λ X : type. λ Y : bool. λ Z : type.  
    <λ a : (X)α.  
      λ b : (if Y then int → int else int)β.  
        λ c : (Z)γ.  
          if a then (c b) else b>
```

$C = \{\alpha = \text{bool}, \gamma = \beta \rightarrow \beta, a' = \beta\}$

```
fun int false (int → int) =  
  <λ a : (int)α.  
    λ b : (if false then int → int else int)β.  
      λ c : (int → int)γ.  
        if a then (c b) else b>
```

$C' = [int/\alpha]C = \{\text{int} = \text{bool}, \gamma = \beta \rightarrow \beta, a' = \beta\}$

# A More Convincing Example

```
let meta fun =  
  λ X : bool.  
    <λ a0 : (if X then int else bool)α.  
      λ a1 : (e1m)β. λ a2 : (e2m)γ. ⋯ λ an : (enm)δ.  
        let x = em in /* Θ(em) = 2n */  
          if a0 then ⋯ else ⋯>
```

$C = \{\alpha = \text{bool}, \dots\}$

```
fun true =  
  <λ a0 : (int)α.  
    λ a1 : (e1m)β. λ a2 : (e2m)γ. ⋯ λ an : (enm)δ.  
      let x = em in /* Θ(em) = 2n */  
        if a0 then ⋯ else ⋯>
```

$C' = [\text{int}/\alpha]C = \{\text{int} = \text{bool}, \dots\}$

# Dealing with Let-Polymorphism

```
let meta fun =  
  λ X : bool.  
    <λ a0 : (if X then int else bool)α.  
      ...>  
in  
~(fun true) 100 /* α = int */ → /* α_1 = int */  
~(fun false) true /* α = bool */ → /* α_2 = bool */
```

# Dealing with Error Messages (future work)

Because of unification based approach, type errors are difficult to be tracked.

Idea: Blame tracking approach<sup>3</sup>.

---

<sup>3</sup>Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the Design Space of Higher-Order Casts. *European Symposium on Programming, 2009*

# Conclusion

(1) Compared with System F style, our approach will *not* preclude well-type expressions of object language.

(2) Compared with C++ templates style, our approach can discover type errors *earlier* and *save compile time* by avoiding unnecessary meta-evaluation.

THANK YOU