Contextual Modal Type Theory
with Polymophic Contexts

Yuito Murase[1], Yuichi Nishiwaki[2], Atsushi Igarashi[1]

1. Kyoto University
2. Unaffiliated

ESOP 2023@Paris

Staged Computation

Staged computation allows programmers to generate code in a structured and type/scope-safe manner [Calcango et al. 2003, Taha et al. 2023]

Applications to

- Compile-time optimization
- Metaprogramming (e.g., procedural macros)

Quasi-quotation to generate code

compile-time stage
run-time stage

```
let x: int code = ‘(1 + y)
  in ‘(2 * $x)
```

Quasi-quotation to generate code

Quote generates code fragment

compile-time stage

run-time stage

```
let x: int code = ‘(1 + y)
  in ‘(2 * $x)
```

Quasi-quotation to generate code

type of code that evaluates to int value

Quote generates code fragment

compile-time stage

run-time stage

```
let x: int code = ‘(1 + y)
  in ‘(2 * $x)
```

Quasi-quotation to generate code

type of code that evaluates to int value

Quote generates code fragment

compile-time stage

run-time stage

```
let x: int code = ‘(1 + y)
   in ‘(2 * $x)
```

Unquote embeds code fragment

Quasi-quotation to generate code

compile-time stage
run-time stage

type of code that evaluates to int value

Quote generates code fragment

```
let x: int code = ‘(1 + y)
  in ‘(2 * $x)
```

Unquote embeds code fragment

```
⇒ ‘(2 * $‘(1 + y))
```

Quasi-quotation to generate code

type of code that evaluates to int value

Quote generates code fragment

compile-time stage

run-time stage

```
let x: int code = ‘(1 + y)
  in ‘(2 * $x)
```

Unquote embeds code fragment

```
⇒ ‘(2 * $‘(1 + y))

⇒ ‘(2 * (1 + y))
```

Type check against quasi-quote

to reject programs that generate ill-typed code

```
let x: int code = ‘(1 + y)
  in ‘(2 * $x)
```

```
let x: str code = ‘(“hello”)
  in ‘(2 * $x)
```

```
let x: int code = ‘(1 + y)
  in ‘(y(2) * $x)
```

Type check against quasi-quote

to reject programs that generate ill-typed code

✓

```
let x: int code = ‘(1 + y)
  in ‘(2 * $x)
```

✗

```
let x: str code = ‘(“hello”)
  in ‘(2 * $x)
```

expected int code,
but got str code

```
let x: int code = ‘(1 + y)
  in ‘(y(2) * $x)
```

Type check against quasi-quote

to reject programs that generate ill-typed code

✓

```
let x: int code = ‘(1 + y)
  in ‘(2 * $x)
```

✗

```
let x: str code = ‘(“hello”)
  in ‘(2 * $x)
```

expected int code,
but got str code

✗

```
let x: int code = ‘(1 + y)
  in ‘(y(2) * $x)
```

No appropriate type for y

Two ways to manage *typing contexts* of code

Implicit-context style [Davies 1996]

```
y:int ⊢ ‘(1 + y): int code
```

maintained in typing judgment

- Applications to real staged programming languages
  [MetaOCaml][Typed Template Haskell][Scala 3 Macros]

Two ways to manage *typing contexts* of code

Implicit-context style [Davies 1996]

```
y:int ⊢ ‘(1 + y): int code
```

maintained in typing judgment

- Applications to real staged programming languages
  [MetaOCaml][Typed Template Haskell][Scala 3 Macros]

Explicit-context style [Nanevski et al. 2008]

```
⊢ ‘<y:int>(1 + y): [int]int code
```

maintained in code types

- Type safety for mutable reference cells & run-time evaluation
  [Rhiger 2012][Kiselyov 2017]
- Applications to proof assistants
  [Pientka et al. 2010]

Explicit-context style is inflexible

Implicit-context style

```
int code
```

Explicit-context style

```
[ ]int code
[int]int code
[int,str→int]int code
…
```

Issue: Inflexible code types

Implicit-context style

```
int code
```

Explicit-context style

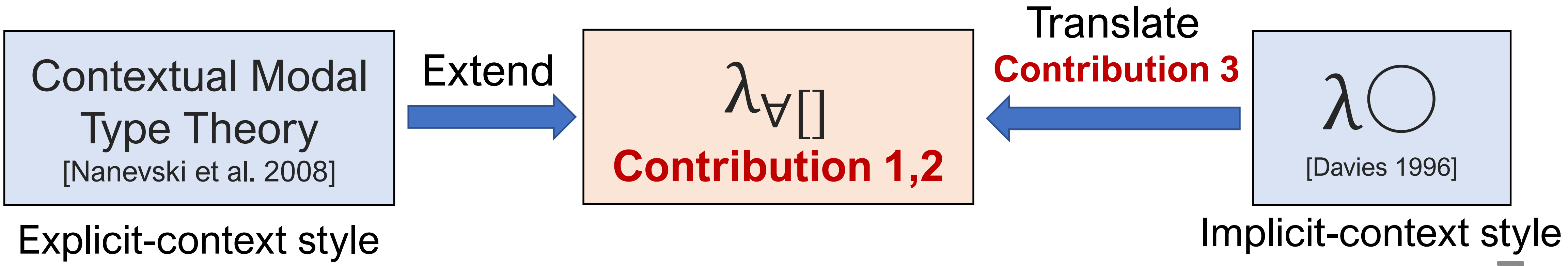
```
[ ]int code
[int]int code
[int,str→int]int code
 …
```

Solution Idea:
Abstraction over contexts

⇒ $\forall\gamma.([\gamma]\texttt{int code})$

Our Contributions

1. Typed lambda calculus $\lambda_{\forall[]}$ with polymorphic contexts
    - Extension of contextual modal type theory
2. Proofs of basic properties of $\lambda_{\forall[]}$
    - Subject Reduction, Strong Normalization and Confluence
3. Type-preserving translation from $\lambda\bigcirc$ to $\lambda_{\forall[]}$



Outline

Fitch-style variant of
contextual modal type theory
[Nanevski et al. 2008]

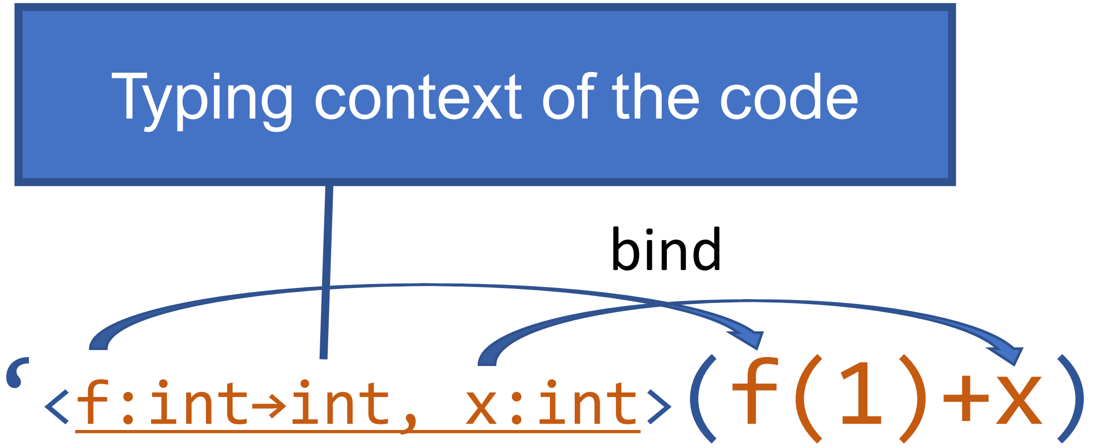
- Introduction
- **$\lambda_{[]}$: Simple Fitch-style contextual modal type theory**
- $\lambda_{\forall[]}$: Polymorphic contexts extension
- Translation from $\lambda\bigcirc$ to $\lambda_{\forall[]}$
- Related work & Conclusion

Quotes and Contextual Modal Types in $\lambda_{[]}$

Typing context of the code

$`_{\langle f:int \to int,\ x:int \rangle}(f(1)+x)$

Quotes and Contextual Modal Types in $\lambda_{[]}$



Quotes and Contextual Modal Types in $\lambda_{[]}$

Typing context of the code

bind

‘<f:int→int, x:int>(f(1)+x)

|| α-equivalent

‘<g:int→int, y:int>(g(1)+y)

Quotes and Contextual Modal Types in $\lambda_{[]}$

Typing context of the code

bind

```
‘<f:int→int, x:int>(f(1)+x)
```

|| α-equivalent

```
‘<g:int→int, y:int>(g(1)+y)
```

```
:[int→int,int ⊢ int]
```

code type with context
(*Contextual Modal Type*)

Quotes and Contextual Modal Types in $\lambda_{[]}$

Typing context of the code

bind

‘<f:int→int, x:int>(f(1)+x)

|| α-equivalent

‘<g:int→int, y:int>(g(1)+y)

:[int→int,int ⊢ int]

Type of the whole expr

code type with context
(*Contextual Modal Type*)

Quotes and Contextual Modal Types in $\lambda_{[]}$

Typing context of the code

bind

```
‘<f:int→int, x:int>(f(1)+x)
```

|| α-equivalent

```
‘<g:int→int, y:int>(g(1)+y)
```

Context of the code (without variable names)

Type of the whole expr

```
:[int→int,int ⊢ int]
```

code type with context (*Contextual Modal Type*)

Unquote in $\lambda_{[]}$ supplies definitions for variables

code of
`[int,int ⊢ int]`

`$(x)`

Unquote in $\lambda_{[]}$ supplies definitions for variables

code of
`[int,int ⊢ int]`

`$(x)[2, 1+y]`

definitions of each
free variables
(*explicit substitution*)

Unquote in $\lambda_{[]}$ supplies definitions for variables

code of
`[int,int ⊦ int]`

`$(x)[2, 1+y]`

definitions of each
free variables
(*explicit substitution*)

Example

`$(‘<x,z:int>(x+z))[2,1+y]`

Unquote in $\lambda_{[]}$ supplies definitions for variables

code of
[int,int ⊦ int]

```
$(x)[2, 1+y]
```

definitions of each free variables (*explicit substitution*)

Example

```
$(‘<x,z:int>(x+z))[2,1+y]
```

Unquote in $\lambda_{[]}$ supplies definitions for variables

code of `[int,int ⊦ int]`

`$(x)[2, 1+y]`

definitions of each free variables (*explicit substitution*)

Example

`$(‘<x,z:int>(x+z))[2,1+y]`

`⇒ (x+z)[x=2,z=1+y]`

Unquote in $\lambda_{[]}$ supplies definitions for variables

code of
[int,int ⊢ int]

$(x)[2, 1+y]

definitions of each
free variables
(*explicit substitution*)

Example

$(‘<x,z:int>(x+z))[2,1+y]

⇒ (x+z)[x=2,z=1+y]

⇒ 2+(1+y)

Example: Sum generation

(in implicit-context style)

```
gen-sum ‘(w) ‘(3 * w)
⇒ ‘(w + 3 * w)
```

Definition

```
let gen-sum x y: [int⊦int]→[int⊦int]→[int⊦int] =
  ‘<z:int>($x[z] + $y[z])
```

Evaluation example

```
gen-sum ‘<w1:int>(w1) ‘<w2:int>(3*w2)
⇒ ‘<z:int>($(‘<w1:int>(w1))[z] + $(‘<w2:int>(3*w2))[z])
⇒ ‘<z:int>(z+3*z)
```

Example: Sum generation

(in implicit-context style)

```
gen-sum ‘(w) ‘(3 * w)
⇒ ‘(w + 3 * w)
```

Definition

```
let gen-sum x y: [int⊦int]→[int⊦int]→[int⊦int] =
  ‘<z:int>($x[z] + $y[z])
```

Evaluation example

```
gen-sum ‘<w1:int>(w1) ‘<w2:int>(3*w2)
⇒ ‘<z:int>($(‘<w1:int>(w1))[z] + $(‘<w2:int>(3*w2))[z])
⇒ ‘<z:int>(z+3*z)
```

Example: Sum generation

(in implicit-context style)

```
gen-sum ‘(w) ‘(3 * w)
⇒ ‘(w + 3 * w)
```

Definition

```
let gen-sum x y: [int⊦int]→[int⊦int]→[int⊦int] =
  ‘<z:int>($x[z] + $y[z])
```

Issue: context of argument code needs to be single variable with int type

Evaluation example

```
gen-sum ‘<w1:int>(w1) ‘<w2:int>(3*w2)
⇒ ‘<z:int>($(‘<w1:int>(w1))[z] + $(‘<w2:int>(3*w2))[z])
⇒ ‘<z:int>(z+3*z)
```

Outline

- Introduction
- $\lambda_{[]}$: Simple Fitch-style contextual modal type theory
- $\boldsymbol{\lambda}_{\forall[]}$: Polymorphic contexts extension
  - Extension to types
  - Extension to terms
  - Example: polymorphic gen-sum
  - Semantics
- Translation from λ○ to $\lambda_{\forall[]}$
- Related work & Conclusion

Context variables to abstract context

In $\lambda_{[]}$

`[⊢ int]`

`[int⊢ int]`

`[str→int,int⊢ int]`

⬇

In $\lambda_{\forall[]}$

$\forall\gamma.[\gamma \vdash \texttt{int}]$

Context variables to abstract context

In $\lambda_{[]}$

```
[⊢ int]
[int⊢ int]
[str→int,int⊢ int]
```

Context variable
(= sequence of types)

In $\lambda_{\forall[]}$

```
∀γ.[γ⊢int]
```

Context variables to abstract context

In $\lambda_{[]}$

`[ ⊢ int]`

`[int ⊢ int]`

`[str→int,int ⊢ int]`

Context variable
(= sequence of types)

In $\lambda_{\forall[]}$

`∀γ.[γ ⊢ int]`

Polymorphic
context type

Context variables to abstract context

In $\lambda_{[]}$

$[\vdash \texttt{int}]$

$[\texttt{int} \vdash \texttt{int}]$

$[\texttt{str}\rightarrow\texttt{int},\texttt{int} \vdash \texttt{int}]$

Context variable
(= sequence of types)

In $\lambda_{\forall[]}$

$\forall\gamma.[\gamma \vdash \texttt{int}]$

Polymorphic context type

Context abstraction $\Lambda\gamma.M$

Context application $M \Leftarrow C$

M:terms

C:seq of types

Series variables to abstract variables

In $\lambda_{[]}$

‘<_>(…)

‘<x:int>(…)

‘<x:str→int,y:int>(…)

↓

In $\lambda_{\forall[]}$

Λγ.‘<🤔:γ>(…)

Series variables to abstract variables

In $\lambda_{[]}$

```
‘<_>(…)
‘<x:int>(…)
‘<x:str→int,y:int>(…)
```

In $\lambda_{\forall[]}$

```
Λγ.‘<ℤ:γ>(…)
```

ℤ: Series variable
(= sequence of variables)

Series variables to abstract variables

In $\lambda_{[]}$

```
‘<_>(…)
‘<x:int>(…)
‘<x:str→int,y:int>(…)
```

In $\lambda_{\forall[]}$

```
Λγ.‘<ℤ:γ>(…)
```

Abstract a part of contexts in quote

ℤ: Series variable
(= sequence of variables)

Series variables as explicit substitutions

code of [γ ⊢ int]

‘<ℤ:γ>($x[🤔])

Series variables as explicit substitutions

code of `[γ ⊢ int]`

`` `<ℤ:γ>($x[ℤ]) ``

ℤ as explicit substitution for free variables of x

Series variables as explicit substitutions

code of `[γ ⊢ int]`

`‘<𝕫:γ>($x[𝕫])`

`𝕫` as explicit substitution for free variables of x

Example

`$(‘<𝕫:γ>($x[𝕫]))[𝕪]`

Series variables as explicit substitutions

code of [γ ⊦ int]

‘<𝕫:γ>($x[𝕫])

𝕫 as explicit substitution for free variables of x

Example

$(‘<𝕫:γ>($x[𝕫]))[𝕪]

Series variables as explicit substitutions

code of [γ ⊢ int]

```
‘<z:γ>($x[z])
```

z as explicit substitution for free variables of x

Example

```
$(‘<z:γ>($x[z]))[y]
```

```
⇒ ($x[z])[z=y]
```

Series variables as explicit substitutions

code of [γ ⊦ int]

‘<𝕫:γ>($x[𝕫])

𝕫 as explicit substitution for free variables of x

Example

$(‘<𝕫:γ>($x[𝕫]))[𝕪]

⇒ ($x[𝕫])[𝕫=𝕪]

⇒ $x[𝕪]

Polymorphic gen-sum

(in implicit-context style)

```
gen-sum ‘(w) ‘(3 * w)
⇒ ‘(w + 3 * w)
```

Monomorphic

```
let gen-sum x y: [int⊢int]→[int⊢int]→[int⊢int] =
  ‘<z:int>($x[z] + $y[z])
```

Polymorphic

```
let poly-gen-sum γ x y: ∀γ.[γ⊢int]→[γ⊢int]→[γ⊢int] =
  ‘<ℤ:γ>($x[ℤ] + $y[ℤ])
```

Context substitution destructs series vars

```
(Λγ.λx[γ⊢int]‘<y:int,ℤ:γ>(y + $x[ℤ]))⇐(int,int)
```

Context substitution destructs series vars

Context Abstraction

Context Application

```
(Λγ.λx[γ⊦int]‘<y:int,ℤ:γ>(y + $x[ℤ]))⇐(int,int)
```

Context substitution destructs series vars

Context Abstraction **Context Application**

```
(Λγ.λx^[γ⊢int]‘<y:int,z:γ>(y + $x[z]))⇐(int,int)

⇒ (λx^[γ⊢int]‘<y:int,z:γ>(y + $x[z]))[γ≔int,int]
```

Context substitution destructs series vars

Context Abstraction

Context Application

```
(Λγ.λx[γ⊢int]‘<y:int,ℤ:γ>(y + $x[ℤ]))⇐(int,int)

⇒ (λx[γ⊢int]‘<y:int,ℤ:γ>(y + $x[ℤ]))[γ≔int,int]
```

Context substitution destructs series vars

Context Abstraction

Context Application

$$(\Lambda\gamma.\lambda x^{[\gamma \vdash \mathrm{int}]}\,`\langle y{:}\mathrm{int}, \mathbb{z}{:}\gamma\rangle(y + \$x[\mathbb{z}])) \Leftarrow (\mathrm{int}, \mathrm{int})$$

$$\Rightarrow\ (\lambda x^{[\gamma \vdash \mathrm{int}]}\,`\langle y{:}\mathrm{int}, \mathbb{z}{:}\gamma\rangle(y + \$x[\mathbb{z}]))[\gamma \coloneqq \mathrm{int}, \mathrm{int}]$$

Context substitution destructs series vars

Context Abstraction

Context Application

$$(\Lambda\gamma.\lambda x^{[\gamma\vdash \mathtt{int}]}\text{‘}\langle y{:}\mathtt{int},\mathbb{z}{:}\gamma\rangle(y + \$x[\mathbb{z}]))\Leftarrow(\mathtt{int},\mathtt{int})$$

$$\Rightarrow (\lambda x^{[\gamma\vdash \mathtt{int}]}\text{‘}\langle y{:}\mathtt{int},\mathbb{z}{:}\gamma\rangle(y + \$x[\mathbb{z}]))[\gamma \coloneqq \mathtt{int},\mathtt{int}]$$

$\mathbb{z}$ destructs to fresh vars a, b

$$\Rightarrow \lambda x^{[\mathtt{int},\mathtt{int}\vdash \mathtt{int}]}\text{‘}\langle y{:}\mathtt{int},a{:}\mathtt{int},b{:}\mathtt{int}\rangle(y + \$x[a,b])$$

Typing Judgment and β-reduction

Typing judgment

(Fitch-style [Clouston 2019])

$$\Gamma \vdash M : A$$

$$\Gamma = \ldots, \hat{\Gamma}_2, 🔒, \hat{\Gamma}_1, 🔒, \hat{\Gamma}_0$$

Level-1 context (compile-time stage)

Level-0 context (run-time stage)

β-reduction

$$M \rightarrow_\beta N$$

$$(\lambda x^A. M)N \rightarrow_\beta M[x := N]_0$$

$$\$({`}_{\langle\hat{\Gamma}\rangle}(M))[\theta] \rightarrow_\beta M[\hat{\Gamma} := \theta]_0$$

$$(\Lambda\gamma. M) \Leftarrow C \rightarrow_\beta M[\gamma := C]$$

Basic Properties

Theorem (Subject Reduction)

If $\Gamma \vdash M : A$ and $M \rightarrow_{\beta} N$, then $\Gamma \vdash N : A$

Theorem (Strong Normalization)

If $\Gamma \vdash M : A$, then M is strongly normalizing

Theorem (Confluence)

If $\Gamma \vdash M : A$, $M \rightarrow_{\beta}^{*} N_1$ and $M \rightarrow_{\beta}^{*} N_2$,

then $\exists N_3. N_1 \rightarrow_{\beta}^{*} N_3$ and $N_2 \rightarrow_{\beta}^{*} N_3$

Proof by Girard's reducibility candidate technique

Outline

- Introduction
- $\lambda_{[]}$: Simple Fitch-style contextual modal type theory
- $\lambda_{\forall[]}$: Polymorphic contexts extension
- Translation from $\lambda\bigcirc$ to $\lambda_{\forall[]}$
  - Motivation
  - Basic idea
  - Formal Properties
- Related work & Conclusion

Why translation from $\lambda\bigcirc$ [Davies 1996] matters

Minimal formulation of implicit-context style calculi

$\lambda\bigcirc$ can be translated to $\lambda_{\forall[]}$

$\lambda_{\forall[]}$ has (at least) minimum capability for staged computation

Because

$\lambda\bigcirc$ provides basis for practical staged programming languages

- MetaOCaml [Kiselyov 2014]
- Typed Template Haskell [Xie et al. 2022]
- Scala 3 macros [Stucki et al. 2021]

Idea: Recover context

In $\lambda\bigcirc$ (Implicit-context style)

```
y:int ⊢ ‘(1 + y): int code
```

Idea: Recover context

In λ◯ (Implicit-context style)

`y:int ⊢ ‘(1 + y): int code` ⇒

Goal: Recover context

- `[ ⊢ int]`
- `[int ⊢ int]`
- `[int,str ⊢ int]`

Idea: Recover context

In λ◯ (Implicit-context style)

y:int ⊢ `(1 + y): int code ⇨

Goal: Recover context

[⊢int]

[int⊢int]

[int,str⊢int]

Method: Use context of the judgment

Translation examples

In $\lambda\bigcirc$ (Implicit-context style)

In $\lambda_{\forall[]}$ (Explicit-context style)

$$y{:}\mathtt{int} \vdash \text{`}(1 + y) : \mathtt{int\ code} \Rightarrow \vdash \text{`}_{\langle y:\mathtt{int}\rangle}(1 + y) : [\mathtt{int} \vdash \mathtt{int}]$$

Translation examples

In $\lambda\bigcirc$ (Implicit-context style)		In $\lambda_{\forall[]}$ (Explicit-context style)
y:int ⊢ ‘(1 + y): int code	➡	⊢ ‘<y:int>(1 + y):[int ⊢ int]
x:int code, y:int ⊢ $x: int	➡	x:[int ⊢ int], y:int ⊢ $x[🤔]: int

Translation examples

In $\lambda\bigcirc$ (Implicit-context style)	In $\lambda_{\forall[]}$ (Explicit-context style)

$$y{:}\mathtt{int} \vdash \text{`}(1 + y) : \mathtt{int\ code} \quad \Rightarrow \quad \vdash \text{`}_{\langle y:\mathtt{int}\rangle}(1 + y) : [\mathtt{int} \vdash \mathtt{int}]$$

$$x{:}\mathtt{int\ code},\ y{:}\mathtt{int} \vdash \$x : \mathtt{int} \quad \Rightarrow \quad x{:}[\mathtt{int} \vdash \mathtt{int}],\ y{:}\mathtt{int} \vdash \$x[y] : \mathtt{int}$$

recover explicit substitutions from typing contexts

Challenge: Mismatching Context

empty context

Context extend with y:int

y:int

$\vdash$ λx^int code^.‘(let y = 1 in $x):int code→int code

Challenge: Mismatching Context

empty context

Context extend with y:int

y:int

$$\vdash \lambda x^{\texttt{int code}}.\,\text{‘}(\texttt{let y = 1 in \$x}) : \texttt{int code} \to \texttt{int code}$$

$$\vdash \lambda x^{[\vdash \texttt{int}]}.\,\text{‘}_{\langle\rangle}(\texttt{let y = 1 in \$(x)[y]}) : [\vdash \texttt{int}] \to [\vdash \texttt{int}]$$

Challenge: Mismatching Context

empty context

Context extend with y:int

y:int

⊢ λx^int code . ‘(let y = 1 in $x):int code→int code

Error: unnecessary explicit subsitution

⊢ λx^[⊢int] . ‘<>(let y = 1 in $(x)[y]):[⊢int]→[⊢int]

Challenge: Mismatching Context

empty context

Context extend with y:int

y:int

```
⊢ λx^{int code}. ‘(let y = 1 in $x):int code→int code
```

Error: unnecessary explicit subsitution

```
⊢ λx^{[ ⊢int]}. ‘_{<>}(let y = 1 in $(x)[y]):[ ⊢int]→[ ⊢int]
```

Solution: Use polymorphic contexts for extended contexts

```
⊢ λx^{∀γ.[γ ⊢int]}. ‘(… $(x⇐int)[y]):(∀γ.[γ ⊢int])→[ ⊢int]
```

Challenge: Mismatching Context

empty context

Context extend with y:int

y:int

⊢ λx^(int code).‘(let y = 1 in $x):int code→int code

Error: unnecessary explicit subsitution

⊢ λx^([⊢int]).‘_(<>)(let y = 1 in $(x)[y]):[⊢int]→[⊢int]

Solution: Use polymorphic contexts for extended contexts

⊢ λx^(∀γ.[γ⊢int]).‘(… $(x⇐int)[y]):(∀γ.[γ⊢int])→[⊢int]

[int⊢int]

Translation is sound

Theorem (Soundness of translation)

We consider two-level fragment of $\lambda\bigcirc$
(Types like int code code does not appear)

Auxiliary object for translating
typing contexts of $\lambda\bigcirc$

If $\Gamma^\circ \vdash_0 M^\circ : A^\circ$ holds in $\lambda\bigcirc$ and $\Gamma^\circ \rightsquigarrow \tilde{\Gamma}$,

$\Rightarrow \left|\tilde{\Gamma}\right|_0 \vdash [\![M^\circ]\!]_{\tilde{\Gamma}} : [\![A^\circ]\!]_{rg(|\tilde{\Gamma}|_1)}$ holds in $\lambda_{\forall[]}$

Translation of terms and types using $\tilde{\Gamma}$

Outline

- Introduction
- $\lambda_{[]}$: Simple Fitch-style contextual modal type theory
- $\lambda_{\forall[]}$: Polymorphic contexts extension
- Translation from $\lambda\bigcirc$ to $\lambda_{\forall[]}$
- Related work & Conclusion

Related work

- Original CMTT [Nanevski et al. 2008]
  - Formalization in dual-context style [Pfenning and Davies 2001, Davies and Pfenning 2001, Kavvos 2017]
  - Syntax: Quote + Meta-variable
  - $\lambda_{[]}$ and $\lambda_{\forall[]}$ are formalized in Fitch-style (or Kripke-style) [Martini and Masini 1996, Davies and Pfenning 2001, Clouston 2019]
  - Both CMTT are extension of S4 modal calculi
- Prior work on polymorphic contexts [Pientka and Dunfield 2008, Puech 2016]
  - $\lambda_{\forall[]}$ allows multiple occurrences of context variables in a context (which prior proposals do not) e.g., `[γ,δ⊢int]`
  - Essential for translation from λ◯

Conclusion

- We proposed $\lambda_{\forall[]}$, a contextual modal type theory with polymorphic contexts
- We proved basic properties $\lambda_{\forall[]}$
- Type-preserving translation from $\lambda\bigcirc$ to $\lambda_{\forall[]}$
  - $\lambda_{\forall[]}$ is capable of applications to staged computation
  - and polymorphic context is essential to the translation

Appendix

Future direction

- Algebraic effects and handlers [Zyuzin 2021] , analytic metaprogramming [Parreaux et al. 2018]
- Improve verbose syntax
  - Approach 1: Inference algorithm
  - Approach 2: Try other representation of contexts e.g., refined environment classifiers [Kiselyov et al. 2017]
- Extension to labelled context, e.g., `[a: int, b: int]`
  - label should be different from variables
  - Need to manage freshness condition, e.g, `a#γ`

Q: Subtyping for translation?

A: We cannot use subtyping [Rhiger 2012] to extend contexts for function of code

[int ⊢ int] → [int ⊢ int]

fails

[int,str ⊢ int] → [int,str ⊢ int]

Translation from $\lambda_{\forall[]}$ to System F

Contexts in $\lambda_{\forall[]}$ implicitly assumes monoid structure, which makes it difficult to translate from $\lambda_{\forall[]}$ to System F

$\lambda_{\forall[]}$ $\forall\gamma.[\gamma,\mathtt{int} \vdash \mathtt{int}]$ $\xrightarrow{\gamma=\mathtt{int},\mathtt{str}}$ $[\mathtt{int},\mathtt{str},\mathtt{int} \vdash \mathtt{int}]$

Translate ↓ Translate ↓

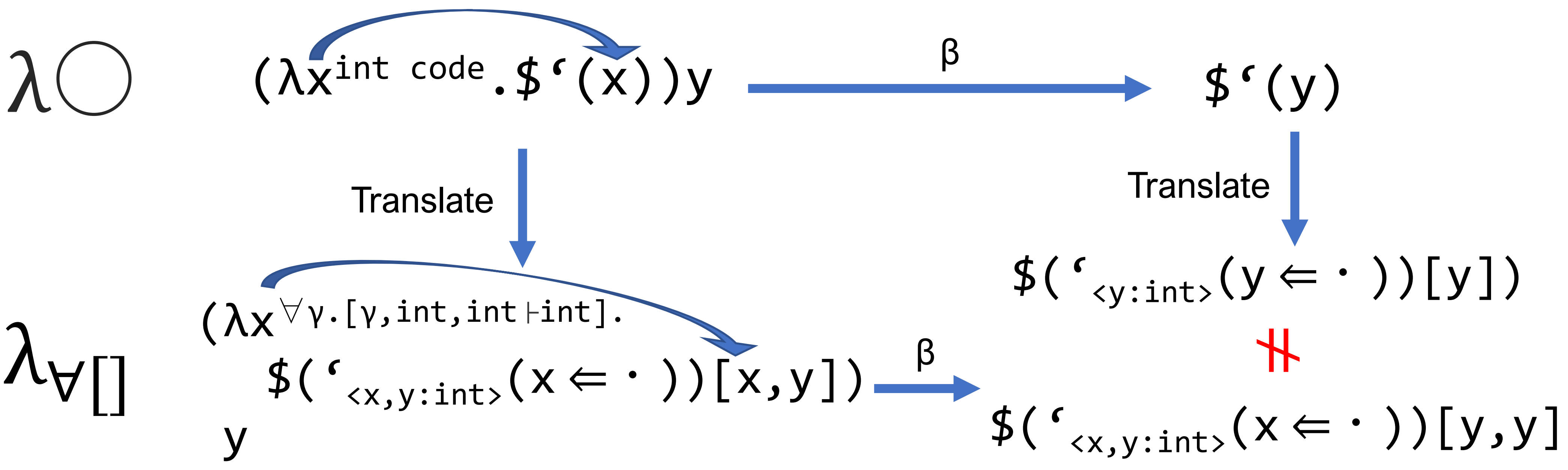
System F $\forall\gamma.(\gamma,(\mathtt{int},()))\to\mathtt{int}$ $(\mathtt{int},(\mathtt{str},(\mathtt{int},())))\to\mathtt{int}$

$\not=$

$\xrightarrow{\gamma=(\mathtt{int},(\mathtt{str},()))}$ $((\mathtt{int},\mathtt{str},()),(\mathtt{int},()))\to\mathtt{int}$

Translation from λ◯ does not preserve reduction steps

The proposed translation does not preserve semantics naively.



Extension with polymorphic types

We can consider several steps of extensions

Step 1: Polymorphic Type $\forall\alpha.[\texttt{int} \vdash \alpha]$

Step 2: Type vars in contextual Modal Types $[\alpha{:}*,\ \texttt{int} \vdash \alpha]$

Step 3: Code of type $A \to \$B[C]$

See: [Jang et al. 2022]

Scope-extrusion with mutable reference cells

Example: Extension with mutable reference cells

```
let r = ref `(1) in
  `(λy:int. ,(r := `(y); `(1)));
  r!
  (* evaluates to `(y) *)
```

y is defined in this scope

y is NOT defined in this scope