Quantitative Types

Syouki Tsuyama

Introduction

Dependent types can refer to values

⇨ Enable advanced verification

e.g. List with length

```
data List : Type -> Nat -> Type where
  . . .

[1, 2] : List Nat 2

head []
```

Type error!

Introduction

Dependently-typed languages easily lead to inefficient programs

```
length : {n : Nat} -> List a n -> Nat
```

Is `n` is used only for the type definition?

Or used for the calculation?

Introduction : Quantitative Types

All variables have multiplicity $(0, 1, \omega)$
Multiplicity indicates how often the variable is used

- 0 : Unused at Runtime ⇨ Can be Erased
- 1 : Used exactly once
- ω : Unristricted

⇨ Needed at runtime

Outline

- Background : A Role of Types
- Background : Dependent Types
- Quantitative Types
- Application Example of Quantitative Types

Outline

- Background : A Role of Types
- Background : Dependent Types
- Quantitative Types
- Application Example of Quantitative Types

Background : Type Systems

Incorrect programs cause bugs

```
print(1 + true)
```

Bug

Type systems can detect bugs before execution

```
print(1 + true)
```

type checker

`1 + true` is wrong!

Outline

- Background : A Role of Types
- Background : Dependent Types
- Quantitative Types
- Application Example of Quantitative Types

Depedent Types

Feature :

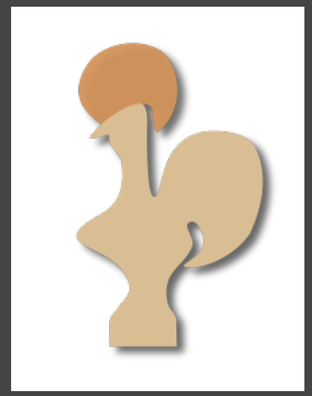
Dependent types can refer to values as well as types

⇨ Allows advanced verification and automatic proof

Dependently-typed languages :

Agda

Idris



Coq

Referring to Types (generics, type parameter)

In plain language, types can refer to types

e.g. Type of List in Scala

Type `List` receives a type

```
trait List[T]
case class Nil[T] extends List[T]
case class Cons[T](head : T, tail : List[T]) extends List[T]

[true] : List[Bool]
[1, 2] : List[Nat]
```

Limit of Verification in Plain Type Systems

The function `head` returns the top element of given list

```
head [1, 2, 3] = 1
head [] = ?
```

`head []` type checks but causes runtime error

`head []` ⇨ type checker ⇨  Bug

Hmm, OK!

Depedent Types

Dependent types can also refer to values

e.g. Type of List with length by Idris

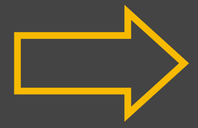
```
data List : Type -> Nat -> Type where
  . . .

[]     : List a 0
[true] : List Bool 1
[1, 2] : List Nat 2
```

Advanced Verification

Recall : In plain language(like scala),

`head []` causes runtime error

```
safe_head : List a (1 + n) -> a
          [] : List a 0
```

`safe-head []`  type checker 

It's wrong!

Outline

- Background : A Role of Types
- Background : Dependent Types
- Quantitative Types
- Application Example of Quantitative Types

Dependent Types lead to Inefficient code

Full Explicit Definition of `safe-head`

```
safe-head : {n : Nat} -> {a : Type} ->
            List a (1 + n) -> a
safe-head {n} {a} (x :: xs) = x
```

Too many arguments

`n` and `a` are needed only for type definition and not relevant to the head calculation

Erasure of type information

```
safe-head : {n : Nat} -> {a : Type} ->
            List a (1 + n) -> a
safe-head {n} {a} (x :: xs) = x
```

⇨ Type Checker ◁ OK!

⇨ Generate more effecient code

```
safe-head (x :: xs) = x
```

No longer need n and a ⇨ Erased

Problem

Difficulty : Distinguish between
necessary and unnecessary variables.

```
length : {n : Nat} -> {a : Type} ->
         List a n -> a
length {n} {a} xs = n
```

`n` is used in the caliculation of length

as well as type definition ⇨ Cannot erase `{n}`

Quantitative Type System

Feature : Trace variable usage

All variables have multiplicity (0, 1, ω)
Multiplicity indicates how often the variable is used

- 0 : Unused at Runtime ⇨ Can be Erased
- 1 : Used exactly once
- ω : Unristricted

⇨ Needed at runtime

Idris2 : Language with Quantative Types

Multiplicity of variables can be specified
at function definitions

a is not used

x is used exactly once

```
id : {0 a : Type} -> (1 x : a) -> a
id {a} x = x
```

Multiplicity 0 : Unused at Runtime

Variable with multiplicity 0

can only be used freely in type definition

```
safe-head : {0 n : Nat} -> {0 a : Type} ->
            List a (1 + n) -> a

safe-head {n} {a} (x :: xs) = x
```

Multiplicity 1 : Linear Usage

Variable with multiplicity 1

must be used exactly once in function body.

```
idNat : (1 x : Nat) -> Nat
idNat x = x
```

Multiplicity ω : Unrestricted Usage

Variable with multiplicity ω

can be used freely anywhere.

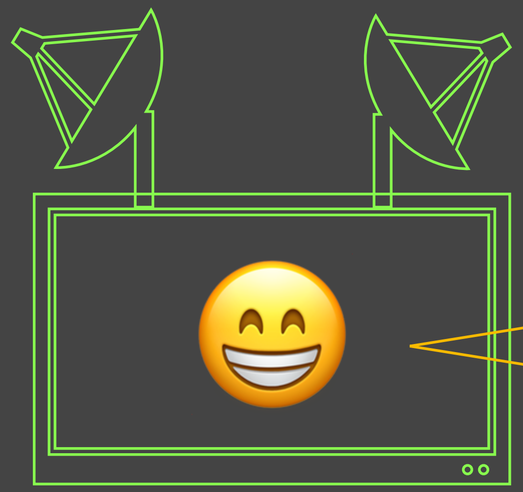
No annotation ⇨ Multiplicity ω

```
length : {n : Nat} -> List a n -> Nat
length {n} _ = n
```

Outline

- Background : A Role of Types
- Background : Dependent Types
- Quantitative Types
- Application Example of Quantitative Types

Application Example : Communication



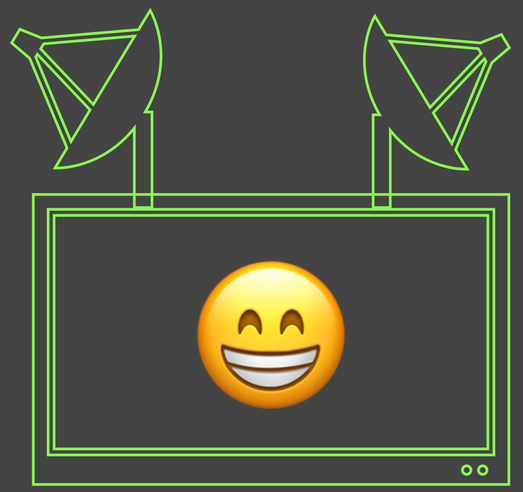
Channel-kun

Hello client! I'm a channel.
You can send one number to the server through me,
and then receive the string returned by the server. OK?

OK. Write his action as

```
{Send Int ⇒ Recv Str ⇒ Close}
```

Dependently-Typed Channel

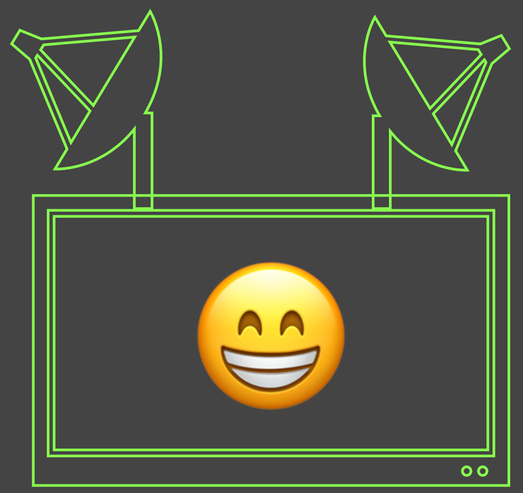


```
data Action : Type where
  Send  : Type -> (next : Action) -> Action
  Recv  : Type -> (next : Action) -> Action
  Close : Action

data Channel : Actions -> Type
```

His type is

```
Channel (Send Int (Recv Str (Close)))
```

Type-safe Channel Operations

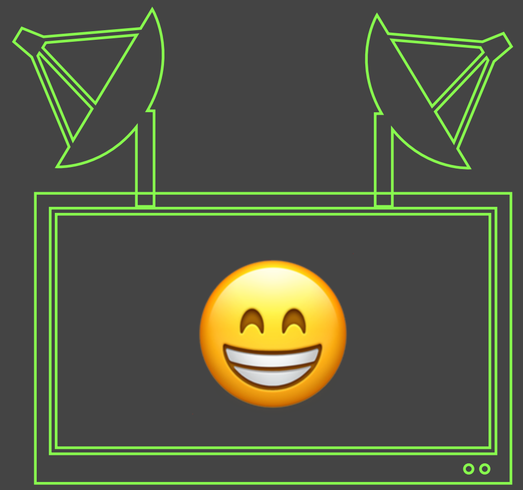


```
send  : (1 chan : Channel (Send ty next))
        -> (val : ty) -> Channel next

recv  : (1 chan : Channel (Recv ty next))
        -> (ty , Channel next)

close : (1 chan : Channel Close) -> ()
```

Apply `send , recv , close`
to him in that order

Type safe operations

Dependent types can detect wrong manipulation.



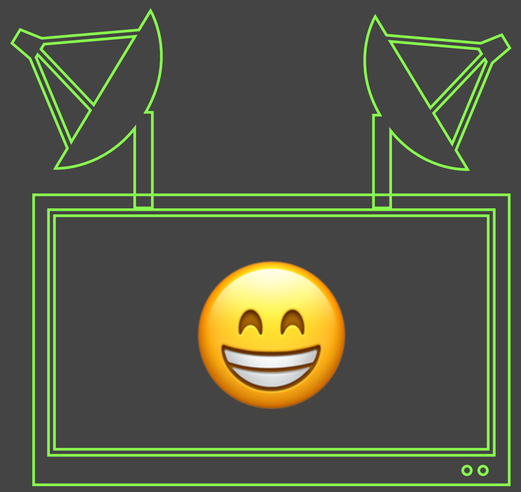
✅ `send 10155 channel_kun`

❌ `recv channel_kun`

❌ `close channel_kun`

```
{Send Int ⇒
 Recv Str ⇒ Close}
```

Correct Program

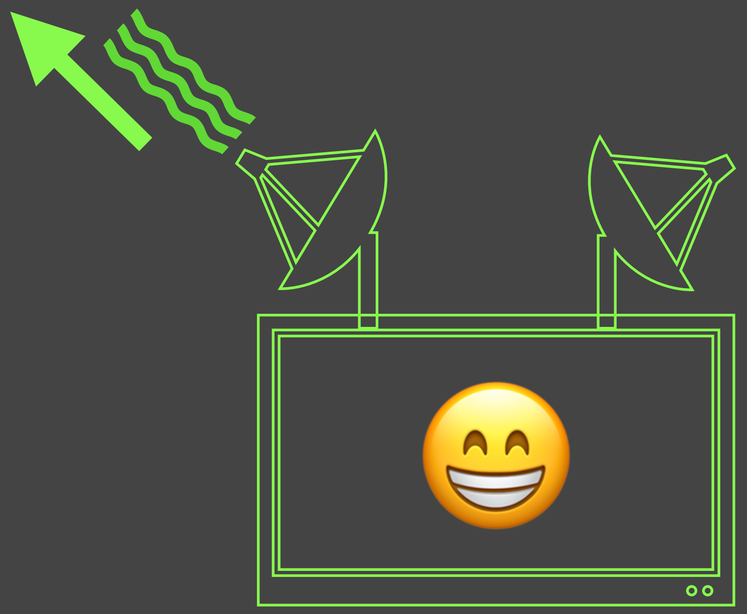


```
let ch1 = send 10155 channel_kun in
let v , ch2 = recv ch1 in
close ch2
```

```
{Send Int ⇒
 Recv Str ⇒ Close}
```

Correct Program

10155



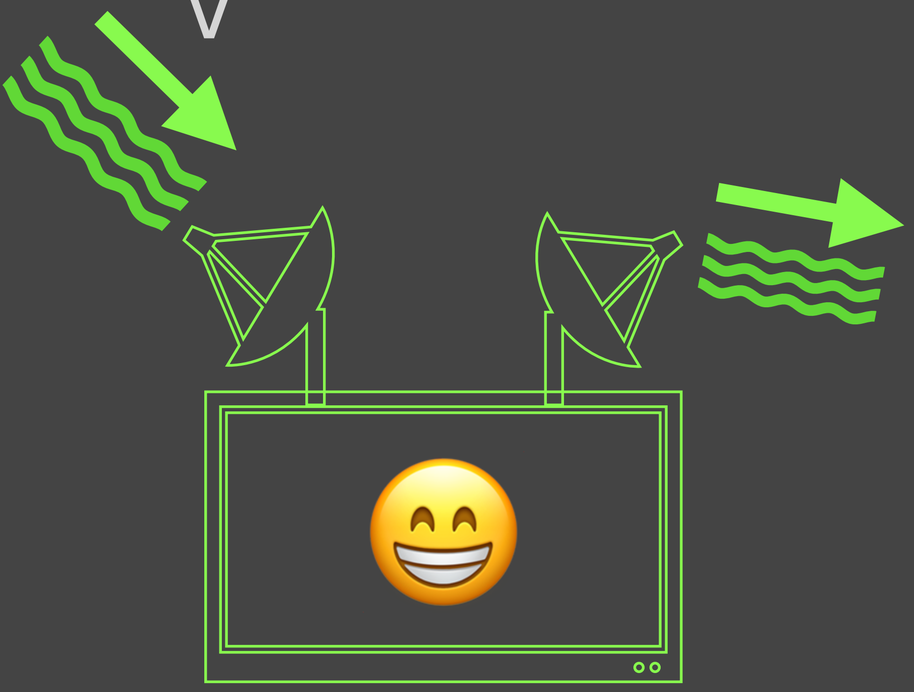
{Send Int ⇒
 Recv Str ⇒ Close}

Send number 10155 to server

```
let ch1 = send 10155 channel_kun in
```

Updated channel_kun
with action {Recv Str ⇒ Close}

Correct Program




```
let ch1 = send 10155 channel_kun in
```

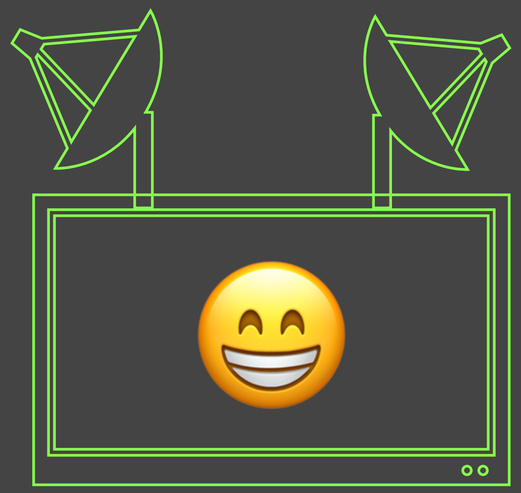
Recieve a value from `ch1`

```
let v , ch2 = recv ch1 in
```

Updated channel_kun
with action {Close}

```
{Send Int ⇒
 Recv Str ⇒ Close}
```

Correct Program



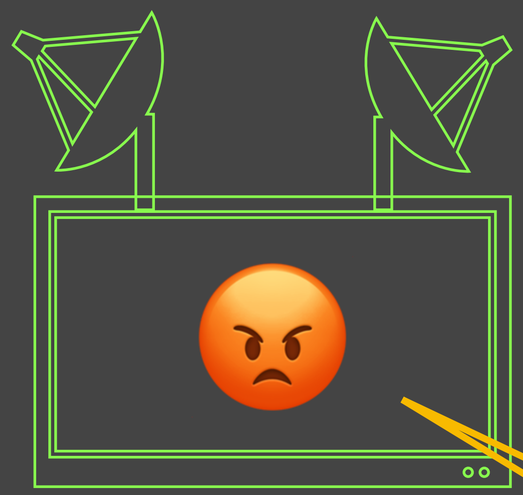
```
let ch1 = send 10155 channel_kun in
let v , ch2 = recv ch’ in
```

Terminate Connection

```
close ch2
```

```
{Send Int ⇒
 Recv Str ⇒ Close}
```

Communication : Incorrect Program



Use send twice to the same channel

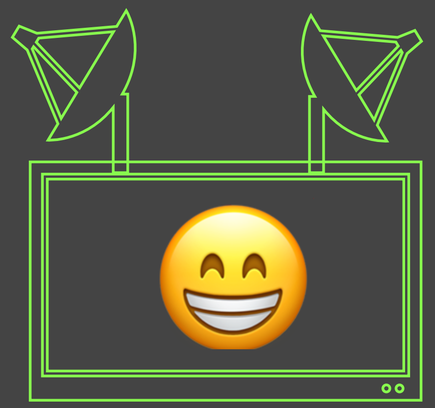
```
let ch’  = send 10155 channel_kun in
let ch’’ = send 30221 channel_kun in
```

You can send **only one** number

Multiplicity of the Channel : 1

Quantitative types can detect duplicate use of channel.

```
main : (1 channel_kun : Chennel (…)) -> ()
main channel_kun =
  let ch’  = send 10155 channel_kun in
  let ch’’ = send 30221 channel_kun in
```



Summary

- Quantitative types make dependently-typed program more efficient
- Quantitative types allow to verify about how often the variable is used

Quantitative Type Theory (QTT)

Linear Types

Feature : Variables must be used exactly once

多くの言語にはない変数の使用状況の制約

Linear Haskell

Illegal Program in Linearly-Typed Language

Linear resource **must be used**

```
add1 : (1 x : Nat) -> Nat
add1 x = 1
```

x must be used in right hand side!

Illegal Program in Linearly-Typed Language

Linear resource **must not be used more than once**

```
add1 : (1 x : Nat) -> Nat
add1 x = x + x
```

x must not be used more than once

in right hand side!