Formalising Session Types ~~Without Worries~~ With Fewer Worries

Wen Kokke, University of Edinburgh

Prologue

Why are we sad?

Why are we sad?

- formalising programming languages is hard
- *shakes fist at the abstract concept of binding* 😤
- lots of tools make it easier (ACMM, Ott, Autosubst, N&K)
- none of those tools work for linear type systems! 😩

- formalising evaluation is tricky
- formalising *concurrent* evaluation is really hard 😭

Prologue

What am I doing?

What am I doing?

I am formalising GV[1]

- a session-typed functional language
- a lambda calculus with channels, send, and receive
- reduction semantics up to structural congruence
- progress, preservation, deadlock-freedom

[1] Wadler, 2014. *Propositions as sessions*

Prologue

What do I want?

I want a formalisation which you can teach to an undergraduate student

Act I.

My Shameful Past

Formalising linearity w/ explicit exchange

```
data _⊢_ : Prectxt → Type → Set where

    var :                        exc : γ ⊢ A    γ ↔ δ
          ---------                    --------------
          ∅ , A ⊢ A                    δ ⊢ A

    ƛ_  : γ , A ⊢ B            _·_ : γ ⊢ A ⊸ B    δ ⊢ A
          ---------                    ------------------
          γ ⊢ A ⊸ B                    γ + δ ⊢ B

-- N.B.
-- Prectxt is a list of types (∅, _,_), _+_ appends
-- lists, and _↔_ is a bijection between lists
```

Formalising linearity w/ explicit exchange

```
data _⊢_ : Prectxt → Type → Set where

    var :                       exc : γ ⊢ A    γ ↔ δ
          ----------                  ---------------
          ∅ , A ⊢ A                   δ ⊢ A

    ƛ_  : γ , A ⊢ B             _·_ : γ ⊢ A ⊸ B    δ ⊢ A
          ----------                  -------------------
          γ ⊢ A ⊸ B                   γ + δ ⊢ B

-- N.B.
-- Prectxt is a list of types (∅, _,_), _+_ appends
-- lists, and _↔_ is a bijection between lists
```

Formalising linearity w/ explicit exchange

bijection

```
data _⊢_ : Prectxt → Type → Set where

  var :                     exc : γ ⊢ A    γ ↔ δ
        ----------                ---------------
        ∅ , A ⊢ A                 δ ⊢ A

  ƛ_  : γ , A ⊢ B           _·_ : γ ⊢ A ⊸ B    δ ⊢ A
        ----------                -------------------
        γ ⊢ A ⊸ B                 γ + δ ⊢ B

-- N.B.
-- Prectxt is a list of types (∅, _,_), _+_ appends
-- lists, and _↔_ is a bijection between lists
```

Formalising linearity w/ explicit exchange

- no variables, no problems, no worries!
- we only have to explicitly manipulate the context!

```
-- what we mean:
swap = λ p → case p of (x,y) → (y,x)

-- what we write:
swap = ƛ (case (exc {...} (pair var var)))
```

var

Formalising linearity w/ explicit exchange

- no variables, no problems, no worries!
- we only have to explicitly manipulate the context!

```
-- what we mean:
swap = λ p → case p of (x,y) → (y,x)

-- what we write:
swap = λ̄ (case (exc {...} (pair var var)))
```

hides about 20 lines of code

Formalising linearity w/ explicit exchange

- understanding terms → understanding implicit context
- explicit exchange → extreme visual clutter
- formalisation of logic w/ explicit structural rules
- no clear correspondence w/ a programming language

Act II.

ACMM[2] and PLFA[3]

[2] Allais, Chapman, McBride, and McKinna. 2017. *Type-and-scope Safe Programs and Their Proofs*

[3] Kokke and Wadler. 2018. *Programming Language Foundations in Agda*

Formalising languages following ACMM

```
data _⊢_ : Prectxt → Type → Set where

    `_   : γ ∋ A
           ------
           γ ⊢ A

    ƛ_   : γ , A ⊢ B      _·_ : γ ⊢ A ⇒ B    γ ⊢ A
           ----------           -------------------
           γ ⊢ A ⇒ B            γ ⊢ B

-- N.B.
-- _∋_ is a de Bruijn index with type info (Z, S_)
```

Formalising languages following ACMM

```
data _⊢_ : Prectxt → Type → Set where

    `_   : γ ∋ A
           ---------
           γ ⊢ A

    ƛ_   : γ , A ⊢ B        _·_ : γ ⊢ A ⇒ B    γ ⊢ A
           ----------             --------------------
           γ ⊢ A ⇒ B              γ ⊢ B

-- N.B.
-- _∋_ is a de Bruijn index with type info (Z, S_)
```

← deBruijn index

Formalising languages following ACMM

```
data _⊢_ : Prectxt → Type → Set where

    `_   : γ ∋ A
           -----
           γ ⊢ A

    ƛ_   : γ , A ⊢ B
           ---------
           γ ⊢ A ⇒ B

    _·_  : γ ⊢ A ⇒ B    γ ⊢ A
           ------------------
           γ ⊢ B

-- N.B.
-- _∋_ is a de Bruijn index with type info (Z, S_)
```

same precontext everywhere

Formalising languages following ACMM

```
data _⊢_ : Prectxt → Type → Set where

    `_   : γ ∋ A
           -----
           γ ⊢ A

    ƛ_   : γ , A ⊢ B        _·_ : γ ⊢ A ⇒ B    γ ⊢ A
           ---------              -----------------
           γ ⊢ A ⇒ B              γ ⊢ B

-- N.B.
-- _∋_ is a de Bruijn index with type info (Z, S_)
```

Formalising languages following ACMM

- no names, but... deBruijn indices, so... worries?
- but at least we have variables now!

```
-- what we mean:
swap = λ p → case p of (x,y) → (y,x)

-- what we write:
swap = ƛ case (` 0) (pair (` 1) (` 0))
```

Formalising languages following ACMM

```
ext     : (∀ {A}   →       γ ∋ A →       δ ∋ A)    -- extend
          ----------------------------------       -- simultaneous
        → (∀ {A B} → γ , B ∋ A → δ , B ∋ A)    -- renaming
                                                   -- ↓
rename  : (∀ {A}   →       γ ∋ A →       δ ∋ A)    -- apply
          ----------------------------------       -- simultaneous
        → (∀ {A}   →       γ ⊢ A →       δ ⊢ A)    -- renaming
                                                   -- ↓
exts    : (∀ {A}   →       γ ∋ A →       δ ⊢ A)    -- extend
          ----------------------------------       -- simultaneous
        → (∀ {A B} → γ , B ∋ A → δ , B ⊢ A)    -- substitution
                                                   -- ↓
subst   : (∀ {A}   →       γ ∋ A →       δ ⊢ A)    -- apply
          ----------------------------------       -- simultaneous
        → (∀ {A}   →       γ ⊢ A →       δ ⊢ A)    -- substitution
```

Formalising languages following ACMM

```
subst : (∀ {A} → γ ∋ A → δ ⊢ A)
        -------------------------
      → (∀ {A} → γ ⊢ A → δ ⊢ A)

subst σ (` k)      =  σ k
subst σ (ƛ N)      =  ƛ (subst (exts σ) N)
subst σ (L · M)    =  (subst σ L) · (subst σ M)
```

Formalising languages following ACMM

***Take-Home Message**:*

Formalisation following ACMM is lightweight and readable.[4]

---

[4] Each proof fits on a slide, and we can teach it to undergraduate students.

Formalising languages following ACMM

```
progress : ∀ {A} → (M : ∅ ⊢ A) → Progress M
progress (` ())                    -- impossible
progress (ƛ N)                     = done V-ƛ
progress (L · M)
  with progress L | progress M
...   | step L—→L′ | _             = step (ξ-·₁ L—→L′)
...   | done V-ƛ   | step M—→M′    = step (ξ-·₂ V-ƛ M—→M′)
...   | done V-ƛ   | done VM       = step (β-ƛ VM)
```

Act III.

Quantitative Type Theory[5]

[5] McBride, 2016. *I Got Plenty o' Nuttin'* & Atkey, 2017. *The Syntax and Semantics of Quantitative Type Theory*

Formalising languages following QTT

- contexts w/ resource annotations
- count resource usage with $\{0, 1, \omega\}$
- contexts parameterised over precontexts on the type level

```
_ : Ctxt (∅ , A , B , C)
_ = ∅ , 1 · A , 0 · B , 0 · C
```

Formalising languages following QTT

```
_ : ∅ , 1 • A , 0 • A ⊸ A ⊢ A
_ = ` S Z

_ : ∅ , 1 • A , 1 • A ⊸ A ⊢ A
_ = (` Z) · (` S Z)

_ : ∅ , ω • A , 1 • A ⊸ A ⊸ A ⊢ A
_ = (` Z) · (` S Z) · (` S Z)
```

Formalising languages following QTT

```
_ : ∅ , 1 • A , 0 • A ⊸ A ⊢ A
_ = ` S Z

_ : ∅ , 1 • A , 1 • A ⊸ A ⊢ A
_ = (` Z) · (` S Z)

_ : ∅ , ω • A , 1 • A ⊸ A ⊸ A ⊢ A
_ = (` Z) · (` S Z) · (` S Z)
```

Formalising languages following QTT

```
_ : ∅ , 1 • A , 0 • A ⊸ A ⊢ A
_ = ` S Z

_ : ∅ , 1 • A , 1 • A ⊸ A ⊢ A
_ = (` Z) · (` S Z)

_ : ∅ , ω • A , 1 • A ⊸ A ⊸ A ⊢ A
_ = (` Z) · (` S Z) · (` S Z)
```

Formalising languages following QTT

```
data _⊢_ : {γ} → Ctxt γ → Type → Set where

  `_  : (x : γ ∋ A)          -- N.B.
        ----------------     -- 1 for x, 0 for each
        identity x ⊢ A       -- other variable in γ

  ƛ_  : Γ , 1 ∙ A ⊢ B       _·_ : Γ ⊢ A ⊸ B    Δ ⊢ A
        ---------------           -----------------
        Γ ⊢ A ⊸ B                 Γ + Δ ⊢ B

-- N.B.
-- Γ and Δ both annotate γ; + is vector addition
```

Formalising languages following QTT

```
data _⊢_ : {y} → Ctxt y → Type → Set where

  `_  : (x : y ∋ A)              -- N.B.
        -----------------        -- 1 for x, 0 for each
        identity x ⊢ A           -- other variable in y

  ƛ_  : Γ , 1 • A ⊢ B            _·_ : Γ ⊢ A ⊸ B    Δ ⊢ A
        ---------------                --------------------
        Γ ⊢ A ⊸ B                      Γ + Δ ⊢ B

-- N.B.
-- Γ and Δ both annotate y; + is vector addition
```

deBruijn index

Formalising languages following QTT

```
data _⊢_ : {y} → Ctxt y → Type → Set where

  `_   : (x : y ∋ A)        -- N.B.
         ----------------   -- 1 for x, 0 for each
         identity x ⊢ A     -- other variable in y

  ƛ_   : Γ , 1 · A ⊢ B      _·_ : Γ ⊢ A ⊸ B    Δ ⊢ A
         ---------------          ------------------
         Γ ⊢ A ⊸ B                Γ + Δ ⊢ B

-- N.B.
-- Γ and Δ both annotate y; + is vector addition
```

Formalising languages following QTT

```
data _⊢_ : {γ} → Ctxt γ → Type → Set where

  `_  : (x : γ ∋ A)         -- N.B.
        ---------------     -- 1 for x, 0 for each
        identity x ⊢ A      -- other variable in γ

  ƛ_  : Γ , 1 · A ⊢ B       _·_ : Γ ⊢ A ⊸ B   Δ ⊢ A
        -------------             -----------------
        Γ ⊢ A ⊸ B                 Γ + Δ ⊢ B

-- N.B.
-- Γ and Δ both annotate γ; + is vector addition
```

Formalising languages following QTT

```
data _⊢_ : {γ} → Ctxt γ → Type → Set where

  `_  : (x : γ ∋ A)            -- N.B.
        ----------------       -- 1 for x, 0 for each
        identity x ⊢ A         -- other variable in γ

  ƛ_  : Γ , 1 ∙ A ⊢ B          _·_ : Γ ⊢ A ⊸ B    Δ ⊢ A
        ---------------              --------------------
        Γ ⊢ A ⊸ B                    Γ + Δ ⊢ B

-- N.B.
-- Γ and Δ both annotate γ; + is vector addition
```

Formalising languages following QTT

```
data _⊢_ : {γ} → Ctxt γ → Type → Set where

  `_  : (x : γ ∋ A)              -- N.B.
        ---------------          -- 1 for x, 0 for each
        identity x ⊢ A           -- other variable in γ

  ƛ_  : Γ , 1 · A ⊢ B           _·_ : Γ ⊢ A ⊸ B    Δ ⊢ A
        --------------                --------------------
        Γ ⊢ A ⊸ B                     Γ + Δ ⊢ B

-- N.B.
-- Γ and Δ both annotate γ; + is vector addition
```

Formalising languages following QTT

Take-Home Message:
Formalisation following QTT is still lightweight and readable.[6]

[6] Each proof fits on a slide, and we can teach it to undergraduate students. They get a little bit sadder than before.

Formalising languages following QTT

```
subst : (σ : ∀ {A} → (x : γ ∋ A) → Ξ x ⊢ A)
      → Γ ⊢ B           -- Ξ is a matrix listing,
        ---------       -- for each variable x,
      → Γ * Ξ ⊢ B       -- the resources used by (σ x)

subst σ (` x)   = rewr lem-` (σ x)
subst σ (ƛ N)   = ƛ (rewr lem-ƛ (subst (exts σ) N))
subst σ (L · M) = rewr lem-· (subst σ L · subst σ M)
```

Formalising languages following QTT

```
subst : (σ : ∀ {A} → (x : γ ∋ A) → Ξ x ⊢ A)

      → Γ ⊢ B          -- Ξ is a matrix listing,
        ----------     -- for each variable x,
      → Γ * Ξ ⊢ B      -- the resources used by (σ x)

subst σ (` x)     = rewr lem-`  (σ x)

subst σ (ƛ N)     = ƛ (rewr lem-ƛ (subst (exts σ) N))

subst σ (L · M) = rewr lem-·  (subst σ L · subst σ M)
```

Problems with using QTT?

- some unrestricted *open* terms are typeable

```
_ : ∅ , ω · A , 1 · A ⊸ A ⊸ A ⊢ A
_ = (` Z) · (` S Z) · (` S Z)
```

- linearity is a global property

```
_ : linear (∅ , A , A ⊸ A) ⊢ A
_ = (` Z) · (` S Z)
```

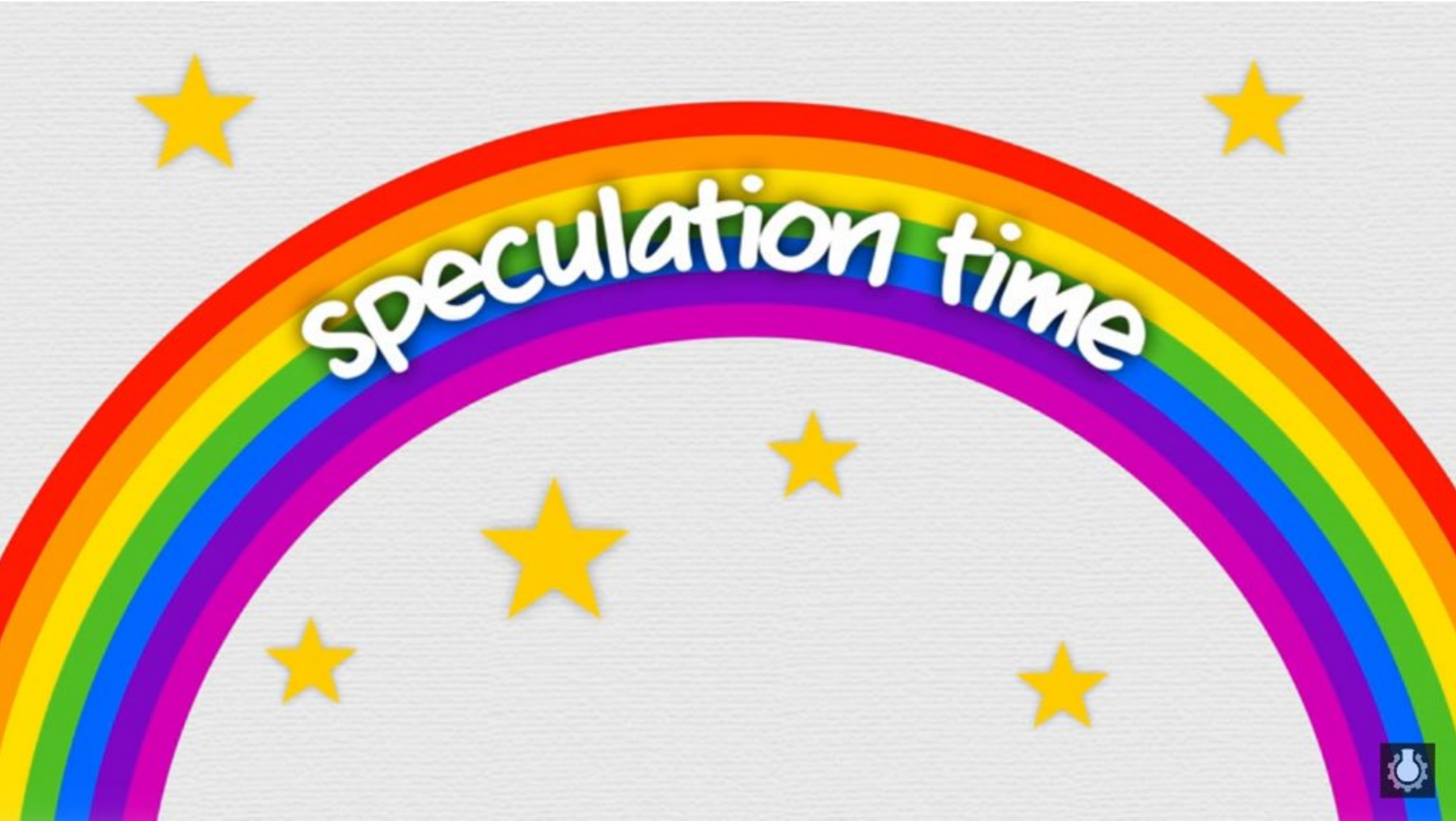
- *true* linearity is a *partial* semiring, as $1 + 1$ is undefined

Conclusions

- formalising programming languages is hard
- formalising *linearly typed* programming languages is harder
- quantitative type theory helps

Act ⟨Bonus⟩.

Formalising concurrent evaluation


speculation time

Formalising concurrent evaluation

Take Home Message:

Encode the invariants you need in your proof in your data types.

Formalising concurrent evaluation

Theorem 1 (Progress).

For every n channels there are $n + 1$ processes trying to act on those channels. There are at most two processes ready to act on any particular channel. When two processes act on the same channel, they do so with opposite behaviours.
Therefore, there is at least one channel on which there are exactly two processes ready to communicate with opposite behaviours.

Formalising concurrent evaluation

Invariants used in proof of progress:

- For every n channels there are $n + 1$ processes trying to act on those channels.
- There are at most two processes ready to act on any particular channel.
- When two processes act on the same channel, they do so with opposite behaviours.

Formalising concurrent evaluation

Definition of configurations

$$C, D ::= \quad \bullet M \quad | \quad \circ M \quad | \quad (\nu x)C \quad | \quad (C \parallel D)$$

Typing rules for configurations

$$\frac{\Gamma, x : S^\sharp \vdash C}{\Gamma \vdash (\nu x)C} \qquad \frac{\Gamma, x : S \vdash C \quad \Delta, x : \neg S \vdash D}{\Gamma, \Delta, x : S^\sharp \vdash (C \parallel D)}$$

Formalising concurrent evaluation

- add channels to our context

```
_ : ∅ , 0 • Send Int End | ∅ , 1 • Int ⊢ Int
_ = ` Z
```

- use vectors to represent configurations

```
Conf Φ Γ = Vec (∃ A . Φ | Γ ⊢ A) (length Φ)
```

- corresponds to $(\nu x_1 \ldots x_n)(P_1 \parallel \cdots \parallel P_{n+1})$
- vectors are sorted by the channel they're ready to act on

Formalising concurrent evaluation

- add channels to our context

```
_ : ∅ , 0 • Send Int End | ∅ , 1 • Int ⊢ Int
_ = ` Z
```

blatant lie

- use vectors to represent configurations

```
Conf Φ Γ = Vec (∃ A . Φ | Γ ⊢ A) (length Φ)
```

- corresponds to $(\nu x_1 \ldots x_n)(P_1 \parallel \cdots \parallel P_{n+1})$
- vectors are sorted by the channel they're ready to act on

Formalising concurrent evaluation

- channels are used in dual ways, so precontexts differ...

```
s : ∅ , 1 • Send u64 End | ∅ ⊢ End
s = ∘ send (chan Z) 1024

r : ∅ , 1 • Recv u64 End | ∅ ⊢ u64
r = • letpair (recv (chan Z))
    $ letunit (wait (` Z)) (` S Z)
```

Formalising concurrent evaluation

- channels are used in dual ways, so precontexts differ…

```
s : ∅ , 1 • Send u64 End | ∅ ⊢ End
s = ∘ send (chan Z) 1024

r : ∅ , 1 • Recv u64 End | ∅ ⊢ u64
r = • letpair (recv (chan Z))
    $ letunit (wait (` Z)) (` S Z)
```

Formalising concurrent evaluation

- count channel usage with integers or $\{-\omega, -1, 0, 1, \omega\}$...

```
s : ∅ , +1 • Send u64 End | ∅ ⊢ End
s = ∘ send (chan⁺ Z) 1024

r : ∅ , -1 • Send u64 End | ∅ ⊢ u64
r = • letpair (recv (chan⁻ Z))
    $ letunit (wait (` Z)) (` S Z)
```

Formalising concurrent evaluation

- count channel usage with integers or $\{-\omega, -1, 0, 1, \omega\}$...

blatant lie

```
s : ∅ , +1 • Send u64 End | ∅ ⊢ End
s = ∘ send (chan^+ Z) 1024

r : ∅ , −1 • Send u64 End | ∅ ⊢ u64
r = • letpair (recv (chan^- Z))
    $ letunit (wait (` Z)) (` S Z)
```

Conclusions

- formalising programming languages is hard
- formalising *linearly typed* programming languages is harder
- formalising concurrent evaluation is *really hard*

- quantitative type theory helps
- we can extend QTT to cover duality (probably)

(Hiya! I'm Wen, and if you'd like to, you can find my stuff at https://wenkokke.github.io 🌈)

Conclusions

- formalising programming languages is hard
- formalising *linearly typed* programming languages is harder
- formalising concurrent evaluation is *really hard*

it me!

- quantitative type theory helps
- we can extend QTT to cover duality (probably)

(Hiya! I'm Wen, and if you'd like to, you can find my stuff at https://wenkokke.github.io 🌈)