Scoped and Typed Staging by Evaluation

Guillaume Allais

University of Strathclyde

(TUPLES,2025)
February 23^{rd} 2025

Table of (Probably Too Much) Contents

Table of Contents

Different motivations

Generic programming e.g. typeclass-based ad-hoc polymorphism

Meta programming e.g. circuit description language

Different motivations

Generic programming e.g. typeclass-based ad-hoc polymorphism

- in a type-safe manner
- with no abstraction cost

Meta programming e.g. circuit description language

- in a type-safe manner
- with no abstraction cost

Different motivations

Generic programming e.g. typeclass-based ad-hoc polymorphism

- using the language itself
- in a type-safe manner
- with no abstraction cost

Meta programming e.g. circuit description language

- in a type-safe manner
- with no abstraction cost

Different motivations

Generic programming e.g. typeclass-based ad-hoc polymorphism

- using the language itself
- in a type-safe manner
- with no abstraction cost

Meta programming e.g. circuit description language

- in a richer language
- in a type-safe manner
- with no abstraction cost

One solution: Two Level Type Theory

A single language equipped with:

One solution: Two Level Type Theory

A single language equipped with:

- two levels: compile time vs. run time

One solution: Two Level Type Theory

A single language equipped with:

- two levels: compile time vs. run time
- compile time functions generate complex run time computations

One solution: Two Level Type Theory

A single language equipped with:

- two levels: compile time vs. run time
- compile time functions generate complex run time computations
- partial evaluation of the compile time fragment

One solution: Two Level Type Theory

A single language equipped with:

- two levels: compile time vs. run time
- compile time functions generate complex run time computations
- partial evaluation of the compile time fragment

	Compile Time	Run Time
Typeclasses	Haskell	Haskell
Circuit Description	STLC	Circuits

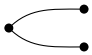
An example: the diagonal of a circuit

Compile time: STLC
Run time: Circuits

An example: the diagonal of a circuit

Compile time: STLC
Run time: Circuits

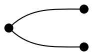
‘dup : ∀[Term *ph* dyn ‘⟨ 1 | 2 ⟩]
‘dup = ‘mix (0 :: 0 :: [])



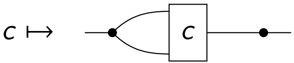
An example: the diagonal of a circuit

Compile time: STLC
Run time: Circuits

```
‘dup : ∀[ Term ph dyn ‘⟨ 1 | 2 ⟩ ]
‘dup = ‘mix (0 :: 0 :: [])
```



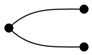
```
‘diag : ∀[ Term src sta (‘⇑ ‘⟨ 2 | 1 ⟩ ‘⇒ ‘⇑ ‘⟨ 1 | 1 ⟩) ]
‘diag = ‘lam ‘⟨ ‘seq ‘dup (‘∼ ‘var here) ⟩
```



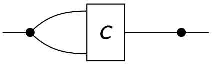
An example: the diagonal of a circuit

Compile time: STLC
Run time: Circuits

```
‘dup : ∀[ Term ph dyn ‘⟨ 1 | 2 ⟩ ]
‘dup = ‘mix (0 :: 0 :: [])
```



```
‘diag : ∀[ Term src sta (‘⇑ ‘⟨ 2 | 1 ⟩ ‘⇒ ‘⇑ ‘⟨ 1 | 1 ⟩) ]
‘diag = ‘lam ‘⟨ ‘seq ‘dup (‘∼ ‘var here) ⟩
```

$c \mapsto$ 


```
‘not : ∀[ Term src dyn ‘⟨ 1 | 1 ⟩ ]
‘not = ‘∼ ‘app ‘diag ‘⟨ ‘nand ⟩
```

‘⟨ 1 | 1 ⟩ ∋ ‘not ⇝ ‘seq ‘dup ‘nand

Table of Contents

Types and Contexts

```
data Type : Set where
  `α    : Type
  _`⇒_ : (A B : Type) → Type
```

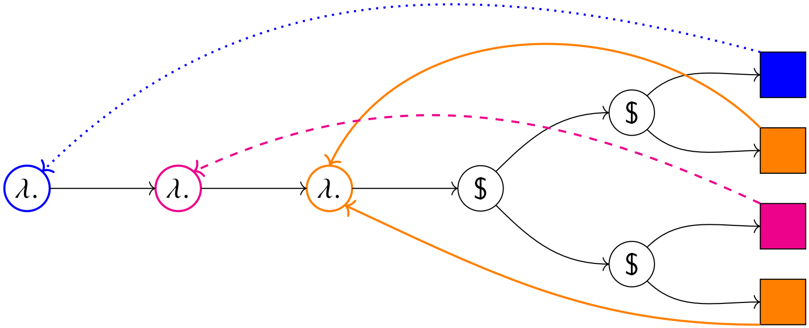
```
variable A B C : Type
```

Types and Contexts

```
data Type : Set where
  ‘α     : Type
  _‘⇒_ : (A B : Type) → Type

variable A B C : Type



data Context : Set where
  ε   : Context
  _,_ : Context → Type → Context

variable Γ Δ Θ : Context
variable P Q : Context → Set
```

De Bruijn indices

The S combinator $(\lambda g.\lambda f.\lambda x.g\,x\,(f\,x))$ in De Bruijn nameless syntax.



Scoped-and-typed De Bruijn indices

```
data Var : Type → Context → Set where
  here  : ∀[               (_, A) ⊢ Var A ]
  there : ∀[ Var A ⇒ (_, B) ⊢ Var A ]
```

$$\frac{}{x : A \vdash x :_v A} \qquad \frac{x :_v A}{y : B \vdash x :_v A}$$

Scoped-and-typed syntax

```
data Term : Type → Context → Set where
```

Scoped-and-typed syntax: variable

$$\text{`var} : \forall[\ \ \mathsf{Var}\ A \Rightarrow \\ \text{- - - - - - - - -} \\ \mathsf{Term}\ A\ ]$$

$$\frac{x :_v A}{x : A}$$

Scoped-and-typed syntax: application

$$\text{`app} : \forall[\ \ \mathsf{Term}\ (A\ \text{`}{\Rightarrow}\ B) \Rightarrow \mathsf{Term}\ A \Rightarrow \ \text{- -}\ \ \mathsf{Term}\ B\ ]$$

$$\frac{f : A \longrightarrow B \qquad t : A}{f\ t : B}$$

Scoped-and-typed syntax: λ-abstraction

$$\text{‘lam} : \forall[\; \frac{(\_, A) \vdash \text{Term } B \Rightarrow}{\text{Term } (A \text{ ‘}\!\Rightarrow B)} \;]$$

$$\frac{x : A \vdash b : B}{\lambda x.b : A \to B}$$

Scoped-and-typed syntax

```
data Term : Type → Context → Set where
  ‘var  : ∀[ Var A ⇒ Term A ]
  ‘app : ∀[ Term (A ‘⇒ B) ⇒ Term A ⇒ Term B ]
  ‘lam : ∀[ (_, A) ⊢ Term B ⇒ Term (A ‘⇒ B) ]
```

```
‘id : ∀[ Term (A ‘⇒ A) ]
‘id = ‘lam (‘var here)
```

Table of Contents

What do we want?

An evaluation function turning terms into values

$\mathsf{eval} : \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Term}\ A\ \Gamma \to \mathsf{Value}\ A\ \Delta$

Category of weakenings

Order Preserving Embeddings (OPEs) can be inductively defined

```
data _≤_ : Context → Context → Set where
  done : ε ≤ ε
  keep : Γ ≤ Δ → Γ , A ≤ Δ , A
  drop : Γ ≤ Δ → Γ     ≤ Δ , A
```

And form a preorder

```
≤-refl : Γ ≤ Γ
≤-trans : Γ ≤ Δ → Δ ≤ Θ → Γ ≤ Θ
```

Action of weakenings on syntax

Well behaved context-indexed families can be transported along OPEs.

$\mathsf{Weaken} : (\mathsf{Context} \rightarrow \mathsf{Set}) \rightarrow \mathsf{Set}$
$\mathsf{Weaken}\ P = \forall\ \{\Gamma\ \Delta\} \rightarrow \Gamma \leq \Delta \rightarrow P\ \Gamma \rightarrow P\ \Delta$

Action of weakenings on syntax

Well behaved context-indexed families can be transported along OPEs.

Weaken : (Context → Set) → Set
Weaken $P = \forall \{\Gamma\ \Delta\} \to \Gamma \leq \Delta \to P\ \Gamma \to P\ \Delta$

In particular, ability to push under binders with:

≤-under : $\Gamma \leq \Gamma\ ,\ A$

Action of weakenings on syntax

Well behaved context-indexed families can be transported along OPEs.

Weaken : (Context → Set) → Set
Weaken $P = \forall \{\Gamma\ \Delta\} \to \Gamma \leq \Delta \to P\ \Gamma \to P\ \Delta$

In particular, ability to push under binders with:

≤-under : $\Gamma \leq \Gamma\ ,\ A$

Some well behaved families: variables, terms

wkVar : Weaken (Var A)
wkTerm : Weaken (Term A)

What do we want (again)?

An evaluation function turning terms into values

$\mathsf{eval} : \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Term}\ A\ \Gamma \to \mathsf{Value}\ A\ \Delta$

What do we want (again)?

An evaluation function turning terms into values

$\mathsf{eval} : \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Term}\ A\ \Gamma \to \mathsf{Value}\ A\ \Delta$

That can run on terms such as (note that g is used in a bigger context)
$\lambda g.\lambda f.\lambda x.g\,x\,(f\,x)$

What do we want (again)?

An evaluation function turning terms into values

$\mathsf{eval} : \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Term}\ A\ \Gamma \to \mathsf{Value}\ A\ \Delta$

That can run on terms such as (note that g is used in a bigger context)
$\lambda g.\lambda f.\lambda x.g\,x\,(f\,x)$

Hence the need for value types that can be weakened!

Model construction: Kripke function spaces

```
record □ (P : Context → Set) (Γ : Context) : Set where
  constructor mk□
  field run□ : ∀[ (Γ ≤_) ⇒ P ]
```

Model construction: Kripke function spaces

```
record □ (P : Context → Set) (Γ : Context) : Set where
  constructor mk□
  field run□ : ∀[ (Γ ≤_) ⇒ P ]
```

```
extract : ∀[ □ P ⇒ P ]
extract p = p .run□ ≤-refl
```

```
duplicate : ∀[ □ P ⇒ □ (□ P) ]
duplicate p .run□ σ .run□ = p .run□ ∘ ≤-trans σ
```

Model construction: Kripke function spaces

```
record □ (P : Context → Set) (Γ : Context) : Set where
  constructor mk□
  field run□ : ∀[ (Γ ≤_) ⇒ P ]
```

```
extract : ∀[ □ P ⇒ P ]
extract p = p .run□ ≤-refl
```

```
duplicate : ∀[ □ P ⇒ □ (□ P) ]
duplicate p .run□ σ .run□ = p .run□ ∘ ≤-trans σ
```

```
Kripke : (P Q : Context → Set) → (Context → Set)
Kripke P Q = □ (P ⇒ Q)
```

```
syntax mk□ (λ σ x → b) = λλ[ σ , x ] b
```

Model construction: Kripke function spaces

```
record □ (P : Context → Set) (Γ : Context) : Set where
  constructor mk□
  field run□ : ∀[ (Γ ≤_) ⇒ P ]

extract : ∀[ □ P ⇒ P ]
extract p = p .run□ ≤-refl

duplicate : ∀[ □ P ⇒ □ (□ P) ]
duplicate p .run□ σ .run□ = p .run□ ∘ ≤-trans σ

Kripke : (P Q : Context → Set) → (Context → Set)
Kripke P Q = □ (P ⇒ Q)

syntax mk□ (λ σ x → b) = λλ[ σ , x ] b

_$$_ : ∀[ Kripke P Q ⇒ (P ⇒ Q) ]
_$$_ = extract

wkKripke : Weaken (Kripke P Q)
wkKripke σ f = duplicate f .run□ σ
```

Model construction: values

$$
\begin{aligned}
&\mathsf{Value} : \mathsf{Type} \to \mathsf{Context} \to \mathsf{Set}\\
&\mathsf{Value}\ `\alpha \quad = \mathsf{Term}\ `\alpha\\
&\mathsf{Value}\ (A\ `\!\!\Rightarrow B) = \mathsf{Kripke}\ (\mathsf{Value}\ A)\ (\mathsf{Value}\ B)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{wkValue} : (A : \mathsf{Type}) \to \mathsf{Weaken}\ (\mathsf{Value}\ A)\\
&\mathsf{wkValue}\ `\alpha \quad \sigma\ v = \mathsf{wkTerm}\ \sigma\ v\\
&\mathsf{wkValue}\ (A\ `\!\!\Rightarrow B)\ \sigma\ v = \mathsf{wkKripke}\ \sigma\ v
\end{aligned}
$$

Model construction: environments

```
record Env (Γ Δ : Context) : Set where
  field get : ∀ {A} → Var A Γ → Value A Δ

extend : ∀[ Env Γ ⇒ □ (Value A ⇒ Env (Γ , A)) ]
extend ρ .run□ σ v .get here       = v
extend ρ .run□ σ v .get (there x) = wkValue _ σ (ρ .get x)
```

Model construction: evaluation

```
eval : Env Γ Δ → Term A Γ → Value A Δ
eval ρ (‘var v)    = ρ .get v
eval ρ (‘app f t) = eval ρ f $$ eval ρ t
eval ρ (‘lam b)   = λλ[ σ , v ] eval (extend ρ .run□ σ v) b
```

Table of Contents

Example

$`\alpha \; `\!\Rightarrow \; `\alpha \ni `\mathsf{app} \; `\mathsf{id}^d \; (`\!\sim \; `\mathsf{app} \; `\mathsf{id}^s \; `\langle \; `\mathsf{id}^d \; \rangle) \leadsto `\mathsf{app} \; `\mathsf{id}^d \; `\mathsf{id}^d$

Phases, Stages, and Types

```
data Phase : Set where
  src stg : Phase

variable ph : Phase
```

Phases, Stages, and Types

```
data Phase : Set where
  src stg : Phase

variable ph : Phase


data Stage : Phase → Set where
  sta : Stage src
  dyn : Stage ph

variable st : Stage ph
```

Phases, Stages, and Types

```
data Phase : Set where
  src stg : Phase

variable ph : Phase


data Stage : Phase → Set where
  sta : Stage src
  dyn : Stage ph

variable st : Stage ph


data Type : Stage ph → Set where
  ‘α     : Type st
  _‘⇒_  : (A B : Type st) → Type st
  ‘⇑_    : Type {src} dyn → Type sta

variable A B C : Type st
```

Scoped-and-typed syntax: the same

```
data Term : (ph : Phase) (st : Stage ph) →
            Type st → Context → Set where
```

Scoped-and-typed syntax: the same

```
data Term : (ph : Phase) (st : Stage ph) →
            Type st → Context → Set where

  ‘var  : ∀[ Var A ⇒ Term ph st A ]
  ‘app : ∀[ Term ph st (A ‘⇒ B) ⇒ Term ph st A ⇒ Term ph st B ]
  ‘lam : ∀[ (_, A) ⊢ Term ph st B ⇒ Term ph st (A ‘⇒ B) ]
```

Scoped-and-typed syntax: but different

```
data Term : (ph : Phase) (st : Stage ph) →
            Type st → Context → Set where

  ‘var  : ∀[ Var A ⇒ Term ph st A ]
  ‘app : ∀[ Term ph st (A ‘⇒ B) ⇒ Term ph st A ⇒ Term ph st B ]
  ‘lam : ∀[ (_, A) ⊢ Term ph st B ⇒ Term ph st (A ‘⇒ B) ]

  ‘⟨_⟩ : ∀[ Term src dyn A ⇒ Term src sta (‘⇑ A) ]
  ‘∼_  : ∀[ Term src sta (‘⇑ A) ⇒ Term src dyn A ]
```

Scoped-and-typed syntax:

```
data Term : (ph : Phase) (st : Stage ph) →
            Type st → Context → Set where

  ‘var  : ∀[ Var A ⇒ Term ph st A ]
  ‘app  : ∀[ Term ph st (A ‘⇒ B) ⇒ Term ph st A ⇒ Term ph st B ]
  ‘lam  : ∀[ (_, A) ⊢ Term ph st B ⇒ Term ph st (A ‘⇒ B) ]

  ‘⟨_⟩ : ∀[ Term src dyn A ⇒ Term src sta (‘⇑ A) ]
  ‘∼_  : ∀[ Term src sta (‘⇑ A) ⇒ Term src dyn A ]
```

```
‘idᵈ : ∀[ Term ph dyn (A ‘⇒ A) ]
‘idᵈ = ‘lam (‘var here)
```

```
‘idˢ : ∀[ Term src sta (A ‘⇒ A) ]
‘idˢ = ‘lam (‘var here)
```

What do we want?

$$\mathsf{eval} : \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Term}\ \mathsf{src}\ \mathit{st}\ A\ \Gamma \to \mathsf{Value}\ \mathit{st}\ A\ \Delta$$

$$\mathsf{stage} : \mathsf{Term}\ \mathsf{src}\ \mathsf{dyn}\ A\ \varepsilon \to \mathsf{Term}\ \mathsf{stg}\ \mathsf{dyn}\ (\mathsf{asStaged}\ A)\ \varepsilon$$

Model construction: values

$$\begin{aligned}
&\mathsf{Value} : (\mathit{st} : \mathsf{Stage}\ \mathsf{src}) \longrightarrow \mathsf{Type}\ \mathit{st} \longrightarrow \mathsf{Context} \longrightarrow \mathsf{Set}\\
&\mathsf{Value}\ \mathsf{sta}\ \ = \mathsf{Static}\\
&\mathsf{Value}\ \mathsf{dyn} = \mathsf{Term}\ \mathsf{stg}\ \mathsf{dyn} \circ \mathsf{asStaged}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Static} : \mathsf{Type}\ \mathsf{sta} \longrightarrow \mathsf{Context} \longrightarrow \mathsf{Set}\\
&\mathsf{Static}\ \text{`}\alpha \qquad\quad\ \ = \mathsf{const}\ \bot\\
&\mathsf{Static}\ (\text{`}\Uparrow A) \quad\ \ = \mathsf{Value}\ \mathsf{dyn}\ A\\
&\mathsf{Static}\ (A\ \text{`}{\Rightarrow}\ B) = \mathsf{Kripke}\ (\mathsf{Static}\ A)\ (\mathsf{Static}\ B)
\end{aligned}$$

Model construction: evaluation

$$
\begin{array}{ll}
\mathsf{eval} : \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Term}\ \mathsf{src}\ st\ A\ \Gamma \to \mathsf{Value}\ st\ A\ \Delta & \\
\mathsf{eval}\ \rho\ (\text{`var}\ v) & = \rho\ .\mathsf{get}\ v \\
\mathsf{eval}\ \rho\ (\text{`app}\ \{st = st\}\ f\ t) & = \mathsf{app}\ st\ (\mathsf{eval}\ \rho\ f)\ (\mathsf{eval}\ \rho\ t) \\
\mathsf{eval}\ \rho\ (\text{`lam}\ \{st = st\}\ b) & = \mathsf{lam}\ st\ (\mathsf{body}\ \rho\ b) \\
\mathsf{eval}\ \rho\ \text{`}\langle\ t\ \rangle & = \mathsf{eval}\ \rho\ t \\
\mathsf{eval}\ \rho\ (\text{`}\sim\ v) & = \mathsf{eval}\ \rho\ v
\end{array}
$$

$$
\begin{array}{l}
\mathsf{body} : \mathsf{Env}\ \Gamma\ \Delta \to \mathsf{Term}\ \mathsf{src}\ st\ B\ (\Gamma\ ,\ A) \to \\
\qquad \mathsf{Kripke}\ (\mathsf{Value}\ st\ A)\ (\mathsf{Value}\ st\ B)\ \Delta \\
\mathsf{body}\ \rho\ b = \lambda\lambda[\ \sigma\ ,\ v\ ]\ \mathsf{eval}\ (\mathsf{extend}\ \rho\ .\mathsf{run}\square\ \sigma\ v)\ b
\end{array}
$$

Model construction: evaluation (ctd)

$$\begin{aligned}
&\mathsf{app} : (st : \mathsf{Stage}\ \mathsf{src})\ \{A\ B : \mathsf{Type}\ st\} \rightarrow \\
&\qquad \mathsf{Value}\ st\ (A\ \text{`}{\Rightarrow}\ B)\ \Gamma \rightarrow \mathsf{Value}\ st\ A\ \Gamma \rightarrow \mathsf{Value}\ st\ B\ \Gamma \\
&\mathsf{app}\ \mathsf{sta}\ \ = _\$\$_ \\
&\mathsf{app}\ \mathsf{dyn} = \text{`}\mathsf{app}
\end{aligned}$$

Model construction: evaluation (ctd)

```
app : (st : Stage src) {A B : Type st} →
      Value st (A ‘⇒ B) Γ → Value st A Γ → Value st B Γ
app sta  = _$$_
app dyn = ‘app
```

```
lam : (st : Stage src) {A B : Type st} →
      Kripke (Value st A) (Value st B) Γ →
      Value st (A ‘⇒ B) Γ
lam sta  b = λλ[ σ , v ] b .run□ σ v
lam dyn b = ‘lam (b .run□ (drop ≤-refl) (‘var here))
```

Model construction: staging

$$\mathsf{stage} : \mathsf{Term}\ \mathsf{src}\ \mathsf{dyn}\ A\ \varepsilon \rightarrow \mathsf{Term}\ \mathsf{stg}\ \mathsf{dyn}\ (\mathsf{asStaged}\ A)\ \varepsilon$$
$$\mathsf{stage} = \mathsf{eval}\ (\lambda\ \mathsf{where}\ .\mathsf{get}\ ())$$

Table of Contents

A circuit language

```
data Type : Stage ph → Set where
  _‘⇒_ : (A B : Type sta) → Type sta
  ‘⇑_   : Type {src} dyn → Type sta
  ‘⟨_|_⟩ : (i o : ℕ) → Type {ph} dyn
```

A circuit language

```
data Type : Stage ph → Set where
  _‘⇒_ : (A B : Type sta) → Type sta
  ‘⇑_   : Type {src} dyn → Type sta
  ‘⟨_|_⟩ : (i o : ℕ) → Type {ph} dyn
```

```
‘nand : ∀[ Term ph dyn ‘⟨ 2 | 1 ⟩ ]
```

A circuit language

```
data Type : Stage ph → Set where
  _‘⇒_ : (A B : Type sta) → Type sta
  ‘⇑_   : Type {src} dyn → Type sta
  ‘⟨_|_⟩ : (i o : ℕ) → Type {ph} dyn
```

```
‘nand : ∀[ Term ph dyn ‘⟨ 2 | 1 ⟩ ]
```

```
‘par : ∀[ Term ph dyn ‘⟨ i₁      | o₁      ⟩ ⇒
          Term ph dyn ‘⟨      i₂ |      o₂ ⟩ ⇒
          Term ph dyn ‘⟨ i₁ + i₂ | o₁ + o₂ ⟩ ]
```

A circuit language

```
data Type : Stage ph → Set where
  _‘⇒_ : (A B : Type sta) → Type sta
  ‘⇑_   : Type {src} dyn → Type sta
  ‘⟨_|_⟩ : (i o : ℕ) → Type {ph} dyn
```

```
‘nand : ∀[ Term ph dyn ‘⟨ 2 | 1 ⟩ ]
```

```
‘par : ∀[ Term ph dyn ‘⟨ i₁      | o₁      ⟩ ⇒
          Term ph dyn ‘⟨      i₂ |      o₂ ⟩ ⇒
          Term ph dyn ‘⟨ i₁ + i₂ | o₁ + o₂ ⟩ ]
```

```
‘seq : ∀[ Term ph dyn ‘⟨ i | m   ⟩ ⇒
          Term ph dyn ‘⟨   m | o ⟩ ⇒
          Term ph dyn ‘⟨ i   | o ⟩ ]
```

A circuit language

```
data Type : Stage ph → Set where
  _‘⇒_ : (A B : Type sta) → Type sta
  ‘⇑_   : Type {src} dyn → Type sta
  ‘⟨_|_⟩ : (i o : ℕ) → Type {ph} dyn
```

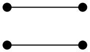
```
‘nand : ∀[ Term ph dyn ‘⟨ 2 | 1 ⟩ ]
```

```
‘par : ∀[ Term ph dyn ‘⟨ i₁       | o₁       ⟩ ⇒
          Term ph dyn ‘⟨       i₂ |       o₂ ⟩ ⇒
          Term ph dyn ‘⟨ i₁ + i₂ | o₁ + o₂ ⟩ ]
```

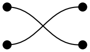
```
‘seq : ∀[ Term ph dyn ‘⟨ i | m     ⟩ ⇒
          Term ph dyn ‘⟨     m | o ⟩ ⇒
          Term ph dyn ‘⟨ i     | o ⟩ ]
```

Wiring examples

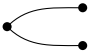
‘id$_2$: ∀[Term *ph* dyn ‘⟨ 2 | 2 ⟩]
‘id$_2$ = ‘mix (0 :: 1 :: [])



‘swap : ∀[Term *ph* dyn ‘⟨ 2 | 2 ⟩]
‘swap = ‘mix (1 :: 0 :: [])



‘dup : ∀[Term *ph* dyn ‘⟨ 1 | 2 ⟩]
‘dup = ‘mix (0 :: 0 :: [])



Recovering the usual logic gates

```
‘diag : ∀[ Term src sta (‘⇑ ‘⟨ 2 | 1 ⟩ ‘⇒ ‘⇑ ‘⟨ 1 | 1 ⟩) ]
‘diag = ‘lam ‘⟨ ‘seq ‘dup (‘~ ‘var here) ⟩

‘not : ∀[ Term src dyn ‘⟨ 1 | 1 ⟩ ]
‘not = ‘~ ‘app ‘diag ‘⟨ ‘nand ⟩

‘and : ∀[ Term src dyn ‘⟨ 2 | 1 ⟩ ]
‘and = ‘seq ‘nand ‘not

‘or : ∀[ Term src dyn ‘⟨ 2 | 1 ⟩ ]
‘or = ‘seq (‘par ‘not ‘not) ‘nand
```

Tabulating a function

```
'tab : ∀[ Term src sta (('Bool '⇒ '⇑ '⟨ 1 | 1 ⟩) '⇒ '⇑ '⟨ 2 | 1 ⟩) ]
'tab = 'lam '⟨ 'seq ('seq ('seq
          ('par 'dup 'dup)
          ('mix (0 :: 2 :: 1 :: 3 :: [])))
          ('par ('seq ('par 'id₁ ('~ 'app ('var here) 'true)) 'and)
                ('seq ('par 'not ('~ 'app ('var here) 'false)) 'and)))
          'or ⟩
```

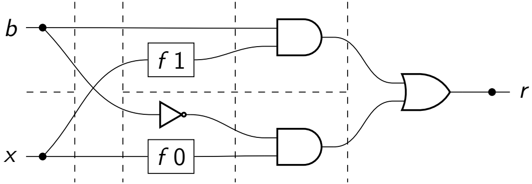
$f \mapsto$ 

Table of Contents

Ongoing and future work

- Soundness and completeness using a logical relation
- Dependently typed circuit description language
- Generic two-level constructions
- Computationally interesting quotes and splices

$$\text{`run} : \forall[\ \text{Term src sta `}\langle\ i \mid o\ \rangle \Rightarrow \text{Term src sta } (\text{`}[\ i\ ]\ \text{`}\!\Rightarrow \text{`}[\ o\ ])\ ]$$
$$\text{`tab} : \forall[\ \text{Term src sta } (\text{`}[\ i\ ]\ \text{`}\!\Rightarrow \text{`}[\ o\ ]) \Rightarrow \text{Term src } st\ \text{`}\langle\ i \mid o\ \rangle\ ]$$

Convention: Implicit context threading

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash f\ t : B}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B}$$

$$\frac{f : A \to B \qquad t : A}{f\ t : B}$$

$$\frac{x : A \vdash b : B}{\lambda x.b : A \to B}$$

Tools: implicit context threading

Combinators:

$$\forall[\_] : (I \to \mathsf{Set}) \to \mathsf{Set}$$
$$\forall[\ F\ ] = \forall\ \{i\} \to F\ i$$

Tools: implicit context threading

Combinators:

$\forall[\_] : (I \to \mathsf{Set}) \to \mathsf{Set}$
$\forall[\ F\ ] = \forall\ \{i\} \to F\ i$

$\_\vdash\_ : (I \to J) \to (J \to \mathsf{Set}) \to (I \to \mathsf{Set})$
$(f \vdash F)\ i = F\ (f\ i)$

Tools: implicit context threading

Combinators:

$\forall[\_] : (I \to \mathsf{Set}) \to \mathsf{Set}$
$\forall[\ F\ ] = \forall\ \{i\} \to F\ i$

$\_\vdash\_ : (I \to J) \to (J \to \mathsf{Set}) \to (I \to \mathsf{Set})$
$(f \vdash F)\ i = F\ (f\ i)$

$\_\Rightarrow\_ : (F\ G : I \to \mathsf{Set}) \to (I \to \mathsf{Set})$
$(F \Rightarrow G)\ i = F\ i \to G\ i$

Tools: implicit context threading

Combinators:

$$\forall[\_] : (I \rightarrow \mathsf{Set}) \rightarrow \mathsf{Set}$$
$$\forall[\ F\ ] = \forall\ \{i\} \rightarrow F\ i$$

$$\_\vdash\_ : (I \rightarrow J) \rightarrow (J \rightarrow \mathsf{Set}) \rightarrow (I \rightarrow \mathsf{Set})$$
$$(f \vdash F)\ i = F\ (f\ i)$$

$$\_\Rightarrow\_ : (F\ G : I \rightarrow \mathsf{Set}) \rightarrow (I \rightarrow \mathsf{Set})$$
$$(F \Rightarrow G)\ i = F\ i \rightarrow G\ i$$

$$\_\cap\_ : (F\ G : I \rightarrow \mathsf{Set}) \rightarrow (I \rightarrow \mathsf{Set})$$
$$(F \cap G)\ i = F\ i \times G\ i$$

Tools: implicit context threading

Combinators:

$$\forall[\_] : (I \rightarrow \mathsf{Set}) \rightarrow \mathsf{Set}$$
$$\forall[\ F\ ] = \forall\ \{i\} \rightarrow F\ i$$

$$\_\vdash\_ : (I \rightarrow J) \rightarrow (J \rightarrow \mathsf{Set}) \rightarrow (I \rightarrow \mathsf{Set})$$
$$(f \vdash F)\ i = F\ (f\ i)$$

$$\_\Rightarrow\_ : (F\ G : I \rightarrow \mathsf{Set}) \rightarrow (I \rightarrow \mathsf{Set})$$
$$(F \Rightarrow G)\ i = F\ i \rightarrow G\ i$$

$$\_\cap\_ : (F\ G : I \rightarrow \mathsf{Set}) \rightarrow (I \rightarrow \mathsf{Set})$$
$$(F \cap G)\ i = F\ i \times G\ i$$

Example:

$$\forall[\ (\_,\ A) \vdash (P \cap Q \Rightarrow Q \cap P)\ ]$$
$$\forall\ \{\Gamma\} \rightarrow (P\ (\Gamma\ ,\ A) \times Q\ (\Gamma\ ,\ A)) \rightarrow (Q\ (\Gamma\ ,\ A) \times P\ (\Gamma\ ,\ A))$$