Scoped and Typed Staging by Evaluation

Guillaume Allais

University of Strathclyde

PLUG
November 8^{th} 2023

Table of Contents

Different motivations

Generic programming:

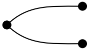
- using the language itself
- in a type-safe manner
- with no abstraction cost

Meta programming:

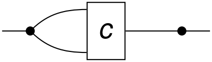
- in a richer language
- in a type-safe manner
- with no abstraction cost

An example: the diagonal of a circuit

```
‘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 ↦ 


```
‘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
```

Convention: Implicit context threading

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

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

Tools: implicit context threading

Combinators:

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

Tools: implicit context threading

Combinators:

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

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

Tools: implicit context threading

Combinators:

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

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

$\_\Rightarrow\_ : (P\ Q : I \to \mathsf{Set}) \to (I \to \mathsf{Set})$
$(P \Rightarrow Q)\ i = P\ i \to Q\ i$

Tools: implicit context threading

Combinators:

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

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

$\_\Rightarrow\_ : (P\ Q : I \rightarrow \mathsf{Set}) \rightarrow (I \rightarrow \mathsf{Set})$
$(P \Rightarrow Q)\ i = P\ i \rightarrow Q\ i$

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

Tools: implicit context threading

Combinators:

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

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

$\_\Rightarrow\_ : (P\ Q : I \to \mathsf{Set}) \to (I \to \mathsf{Set})$
$(P \Rightarrow Q)\ i = P\ i \to Q\ i$

$\_\cap\_ : (P\ Q : I \to \mathsf{Set}) \to (I \to \mathsf{Set})$
$(P \cap Q)\ i = P\ i \times Q\ i$

Example:

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

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

‘var : ∀[Var A ⇒
- - - - - - - - -
Term A]

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

Scoped-and-typed syntax: application

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

$$\dfrac{f : A \to B \qquad t : A}{f\,t : B}$$

Scoped-and-typed syntax: λ-abstraction

$$\text{`lam} : \forall[\ \dfrac{(\_, A) \vdash \text{Term } B \Rightarrow}{\text{Term } (A \text{ `}{\Rightarrow}\ B)}\ ]$$

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

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?

$$\text{eval} : \text{Env}\ \Gamma\ \Delta \rightarrow \text{Term}\ A\ \Gamma \rightarrow \text{Value}\ A\ \Delta$$

Category of weakenings

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

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

Action of weakenings on syntax

```
Weaken : (Context → Set) → Set
Weaken P = ∀ {Γ Δ} → Γ ≤ Δ → P Γ → P Δ

wkVar : Weaken (Var A)
wkVar (drop σ)  v          = there (wkVar σ v)
wkVar (keep σ) here        = here
wkVar (keep σ) (there v) = there (wkVar σ v)

wkTerm : Weaken (Term A)
wkTerm σ (‘var v)    = ‘var (wkVar σ v)
wkTerm σ (‘app f t) = ‘app (wkTerm σ f) (wkTerm σ t)
wkTerm σ (‘lam b)   = ‘lam (wkTerm (keep σ) b)
```

Model construction: Kripke function spaces

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

Model construction: Kripke function spaces

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

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

extract : ∀[ □ P ⇒ P ]              duplicate : ∀[ □ P ⇒ □ (□ P) ]
extract p = p .run□ ≤-refl          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 □ (A : Context → Set) (Γ : Context) : Set where
  constructor mk□
  field run□ : ∀[ (Γ ≤_) ⇒ A ]
```

```
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

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

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

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

$`\alpha \; `\Rightarrow \; `\alpha \ni$ ‘app ‘id^d (‘$\sim$ ‘app ‘id^s ‘$\langle$ ‘id^d $\rangle$) $\rightsquigarrow$ ‘app ‘id^d ‘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

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

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) ]
```

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 ]
```

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

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

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

Model construction: evaluation

```
eval : Env Γ Δ → Term src st A Γ → Value st A Δ
eval ρ (‘var v)            = ρ .get v
eval ρ (‘app {st = st} f t) = app st (eval ρ f) (eval ρ t)
eval ρ (‘lam {st = st} b)   = lam st (body ρ b)
eval ρ ‘⟨ t ⟩               = eval ρ t
eval ρ (‘∼ v)              = eval ρ v

body : Env Γ Δ → Term src st B (Γ , A) →
       Kripke (Value st A) (Value st B) Δ
body ρ b = λλ[ σ , v ] eval (extend ρ .run□ σ v) b
```

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
```

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

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

$\mathsf{stage} = \mathsf{eval}\ (\lambda\ \mathsf{where}\ .\mathsf{get}\ ())$

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

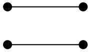
```
data Type : Stage ph → Set where
  _‘⇒_ : (A B : Type sta) → Type sta
  ‘⇑_   : Type {src} dyn → Type sta
  ‘⟨_|_⟩ : (i o : ℕ) → Type {ph} dyn


‘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

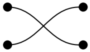
```
data Type : Stage ph → Set where
  _‘⇒_ : (A B : Type sta) → Type sta
  ‘⇑_  : Type {src} dyn → Type sta
  ‘⟨_|_⟩ : (i o : ℕ) → Type {ph} dyn


‘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 ⟩ ]


‘mix : Vec (Fin i) o → ∀[ Term ph dyn ‘⟨ i | o ⟩ ]
```

Wiring examples

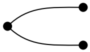
‘id_2 : ∀[Term *ph* dyn ‘⟨ 2 | 2 ⟩]
‘id_2 = ‘mix (0 :: 1 :: [])



‘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 ⟩
```

$f \mapsto$

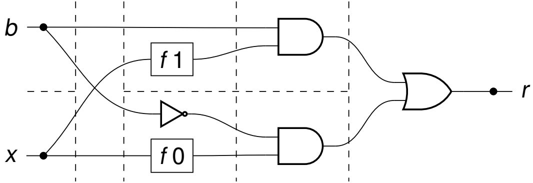


Table of Contents

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

  ‘run : ∀[Term src sta ‘⟨ *i* | *o* ⟩ ⇒ Term src sta (‘[*i*] ‘⇒ ‘[*o*])]
  ‘tab : ∀[Term src sta (‘[*i*] ‘⇒ ‘[*o*]) ⇒ Term src *st* ‘⟨ *i* | *o* ⟩]