

Foundations of Software Fall 2019

Week 9

Different Kinds of Maps

What is missing?

$Term \rightarrow Term \quad (\lambda x.t)$
 $Type \rightarrow Term \quad (\Lambda X.t)$

Different Kinds of Maps

What is missing?

| | | | |
|-------------|---|-------------|-----------------|
| <i>Term</i> | → | <i>Term</i> | $(\lambda x.t)$ |
| <i>Type</i> | → | <i>Term</i> | $(\Lambda X.t)$ |
| <i>Type</i> | → | <i>Type</i> | ??? |
| <i>Term</i> | → | <i>Type</i> | ??? |

Agenda today:

- ▶ Type operators
- ▶ Dependent types

Type Operators and System F_ω

Type Operators

Example. Type operators in Scala:

```
type MkFun[T] = T => T  
val f: MkFun[Int] = (x: Int) => x
```

Type Operators

Example. Type operators in Scala:

```
type MkFun[T] = T => T  
val f: MkFun[Int] = (x: Int) => x
```

Type operators are functions at type-level.

$$\lambda X :: K.T$$

Type Operators

Example. Type operators in Scala:

```
type MkFun[T] = T => T  
val f: MkFun[Int] = (x: Int) => x
```

Type operators are functions at type-level.

$\lambda X :: K.T$

Two Problems:

- ▶ Type checking of type operators
- ▶ Equivalence of types

Kinding

Problem: avoid meaningless types, like *MkFun[Int, String]*.

Kinding

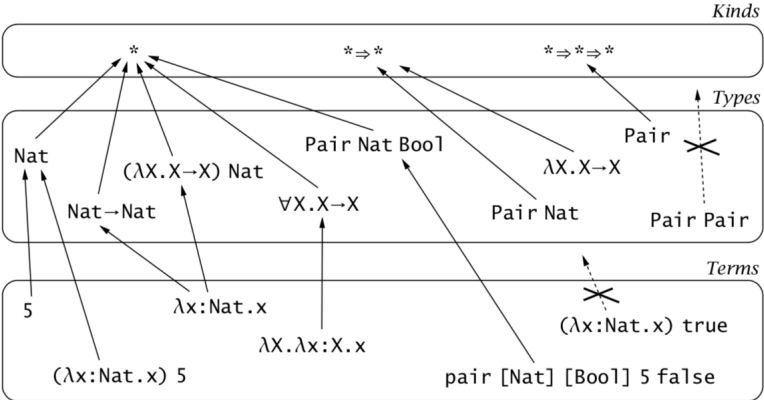
Problem: avoid meaningless types, like *MkFun[Int, String]*.

- * proper types, e.g. *Bool*, *Int* \rightarrow *Int*
- * \Rightarrow * type operators: map proper type to proper type
- * \Rightarrow * \Rightarrow * two-argument operators
- (* \Rightarrow *) \Rightarrow * type operators: map type operators to proper types

Kinding

Problem: avoid meaningless types, like *MkFun[Int, String]*.

- * proper types, e.g. *Bool*, *Int → Int*
- * ⇒ * type operators: map proper type to proper type
- * ⇒ * ⇒ * two-argument operators
- (* ⇒ *) ⇒ * type operators: map type operators to proper types



Equivalence of Types

Problem: all the types below are equivalent

$$\begin{array}{lll} \text{Nat} \rightarrow \text{Bool} & \text{Nat} \rightarrow \text{Id Bool} & \text{Id Nat} \rightarrow \text{Id Bool} \\ \text{Id Nat} \rightarrow \text{Bool} & \text{Id}(\text{Nat} \rightarrow \text{Bool}) & \text{Id}(\text{Id}(\text{Id Nat} \rightarrow \text{Bool})) \end{array}$$

We need to introduce *definitional equivalence* relation on types, written $S \equiv T$. The most important rule is:

$$(\lambda X :: K.S) T \equiv [X \mapsto T]S \quad (\text{Q-APPABS})$$

And we need one typing rule:

$$\frac{\Gamma \vdash t : S \quad S \equiv T}{\Gamma \vdash t : T} \quad (\text{T-EQ})$$

First-class Type Operators

Scala supports passing type operators as argument:

```
def makeInt[F[_]](f: () => F[Int]): F[Int] = f()
```

```
makeInt[List]() => List[Int](3)
```

```
makeInt[Option]() => None
```

First-class type operators supports *polymorphism* for type operators, which enables more patterns in type-safe functional programming.

System F_ω

Formalizing first-class type operators leads to System F_ω :

| | |
|--------------------|-----------------------------|
| $t ::= \dots$ | <i>terms</i> |
| $\lambda X :: K.t$ | <i>type abstraction</i> |
| $T ::=$ | <i>types</i> |
| X | <i>type variable</i> |
| $T \rightarrow T$ | <i>type of functions</i> |
| $\forall X :: K.T$ | <i>universal type</i> |
| $\lambda X :: K.T$ | <i>operator abstraction</i> |
| $T T$ | <i>operator application</i> |
| $K ::=$ | <i>kinds</i> |
| $*$ | <i>kind of proper types</i> |
| $K \Rightarrow K$ | <i>kind of operators</i> |

Dependent Types

Why Does It Matter?

Example 1. Track length of vectors in types:

$$\mathit{Vector} \quad :: \quad \mathit{Nat} \rightarrow *$$
$$\mathit{first} \quad : \quad (n:\mathit{Nat}) \rightarrow \mathit{Vector} (n + 1) \rightarrow D$$

$(x:S) \rightarrow T$ is called **dependent function type**. It is impossible to pass a vector of length 0 to the function *first*.

Why Does It Matter?

Example 1. Track length of vectors in types:

$$\begin{aligned} \text{Vector} &:: \text{Nat} \rightarrow * \\ \text{first} &: (n:\text{Nat}) \rightarrow \text{Vector } (n + 1) \rightarrow D \end{aligned}$$

$(x:S) \rightarrow T$ is called **dependent function type**. It is impossible to pass a vector of length 0 to the function *first*.

Example 2. Safe formatting for *sprintf*:

$$\text{sprintf} \quad : \quad (f:\text{Format}) \rightarrow \text{Data}(f) \rightarrow \text{String}$$
$$\begin{aligned} \text{Data}([]) &= \text{Unit} \\ \text{Data}("%d" :: cs) &= \text{Nat} * \text{Data}(cs) \\ \text{Data}("%s" :: cs) &= \text{String} * \text{Data}(cs) \\ \text{Data}(c :: cs) &= \text{Data}(cs) \end{aligned}$$

Dependent Function Type (a.k.a. Π Types)

A dependent function type is inhabited by *a dependent function*:

$$\lambda x:S.t \quad : \quad (x:S) \rightarrow T$$

Dependent Function Type (a.k.a. Π Types)

A dependent function type is inhabited by a *dependent function*:

$$\lambda x:S.t \quad : \quad (x:S) \rightarrow T$$

If T does not depend on x , it degenerates to function types:

$$(x:S) \rightarrow T = S \rightarrow T \quad \text{where } x \text{ does not appear free in } T$$

Dependent Function Type (a.k.a. Π Types)

A dependent function type is inhabited by a *dependent function*:

$$\lambda x:S.t \quad : \quad (x:S) \rightarrow T$$

If T does not depend on x , it degenerates to function types:

$$(x:S) \rightarrow T = S \rightarrow T \quad \text{where } x \text{ does not appear free in } T$$

By Curry-Howard correspondence, it corresponds to universal quantification:

$$(x:A) \rightarrow B(x) \quad \longleftrightarrow \quad \forall x:A.B(x)$$

First-Order Dependent Types

First-Order Dependent Types: λLF

System λLF generalizes STLC with dependent function types and *type families*.

| | |
|-----------------------|--------------------------------|
| $t ::=$ | <i>terms</i> |
| x | <i>variable</i> |
| $\lambda x:T.t$ | <i>abstraction</i> |
| $t\ t$ | <i>application</i> |
| $T ::=$ | <i>types</i> |
| X | <i>type/family variable</i> |
| $(x:T) \rightarrow T$ | <i>dependent function type</i> |
| $T\ t$ | <i>type family application</i> |

Type or family variables X can only be declared in the typing context Γ . E.g., we may assume $\mathit{Vector} :: \mathit{Nat} \rightarrow *$ as a type family variable.

System λLF : Kinds

Kinds can distinguish *proper types* from *type families*.

$K ::=$ *kinds*
* *kinds of proper types*
 $(x:T) \rightarrow K$ *kind of type families*

$\Gamma ::=$ *contexts*
 \emptyset *empty context*
 $\Gamma, x:T$ *term variable binding*
 $\Gamma, X::K$ *type variable binding*

Well-formed kinds

$\Gamma \vdash K$

$\Gamma \vdash *$ (WF-STAR)

$$\frac{\Gamma \vdash T :: * \quad \Gamma, x:T \vdash K}{\Gamma \vdash (x:T) \rightarrow K}$$
 (WF-PI)

System λLF : Kinding

Kinding ensures that types are well-formed

$\Gamma \vdash T :: K$

$$\frac{X :: K \in \Gamma \quad \Gamma \vdash K}{\Gamma \vdash X :: K} \quad (\text{K-VAR})$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x:T_1 \vdash T_2 :: *}{\Gamma \vdash (x:T_1) \rightarrow T_2 :: *} \quad (\text{K-PI})$$

$$\frac{\Gamma \vdash S :: (x:T) \rightarrow K \quad \Gamma \vdash t : T}{\Gamma \vdash S t :: [x \mapsto t]K} \quad (\text{K-APP})$$

$$\frac{\Gamma \vdash T :: K \quad \Gamma \vdash K \equiv K'}{\Gamma \vdash T :: K'} \quad (\text{K-CONV})$$

System λLF : Typing

Typing ensures that terms are well-formed

$\Gamma \vdash t :: T$

$$\frac{x:T \in \Gamma \quad \Gamma \vdash T :: *}{\Gamma \vdash x : T}$$

(T-VAR)

$$\frac{\Gamma \vdash S :: * \quad \Gamma, x:S \vdash t : T}{\Gamma \vdash \lambda x:S.t : (x:S) \rightarrow T}$$

(T-ABS)

$$\frac{\Gamma \vdash t_1 : (x:S) \rightarrow T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1 t_2 :: [x \mapsto t_2] T}$$

(T-APP)

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T \equiv T' :: *}{\Gamma \vdash t : T'}$$

(T-CONV)

System λLF : Equivalence Rules

With types in kinds, and terms in types, equivalence becomes more complex than System F_ω .

$$\text{Vector } ((\lambda n:\mathbb{N}.n * n) 2) \leftrightarrow \text{Vector } 4$$

λLF defines on several equivalence relations:

- ▶ kind equivalence $\Gamma \vdash K \equiv K'$
- ▶ type equivalence $\Gamma \vdash T \equiv T' :: *$
- ▶ term equivalence $\Gamma \vdash t \equiv t' : T$

System λLF : Equivalence Rules

With types in kinds, and terms in types, equivalence becomes more complex than System F_ω .

$$\text{Vector } ((\lambda n:\mathbb{N}.n * n) 2) \leftrightarrow \text{Vector } 4$$

λLF defines on several equivalence relations:

- ▶ kind equivalence $\Gamma \vdash K \equiv K'$
- ▶ type equivalence $\Gamma \vdash T \equiv T' :: *$
- ▶ term equivalence $\Gamma \vdash t \equiv t' : T$

For *decidable* type checking, type systems usually embrace

- ▶ **definitional equality**, i.e. equality by definition (e.g. $x := 3$)
- ▶ **computational equality**, usually β -equality and η -equality.

System λLF : Kind Equivalence

$$\frac{\Gamma \vdash T_1 \equiv T_2 :: * \quad \Gamma, x:T_1 \vdash K_1 \equiv K_2}{\Gamma \vdash (x:T_1) \rightarrow K_1 \equiv (x:T_2) \rightarrow K_2} \quad (\text{QK-PI})$$

$$\frac{\Gamma \vdash K}{\Gamma \vdash K \equiv K} \quad (\text{QK-REFL})$$

$$\frac{\Gamma \vdash K_1 \equiv K_2}{\Gamma \vdash K_2 \equiv K_1} \quad (\text{QK-SYM})$$

$$\frac{\Gamma \vdash K_1 \equiv K_2 \quad \Gamma \vdash K_2 \equiv K_3}{\Gamma \vdash K_1 \equiv K_3} \quad (\text{QK-TRANS})$$

System λLF : Type Equivalence

$$\frac{\Gamma \vdash S_1 \equiv T_1 :: * \quad \Gamma, x:T_1 \vdash S_2 \equiv T_2 :: *}{\Gamma \vdash (x:S_1) \rightarrow S_2 \equiv (x:T_1) \rightarrow T_2 :: *} \quad (\text{QT-PI})$$

$$\frac{\Gamma \vdash S_1 \equiv S_2 :: (x:T) \rightarrow K \quad \Gamma \vdash t_1 \equiv t_2 : T}{\Gamma \vdash S_1 t_1 \equiv S_2 t_2 :: [x \mapsto t_1]K} \quad (\text{QT-APP})$$

$$\frac{\Gamma \vdash T :: K}{\Gamma \vdash T \equiv T :: K} \quad (\text{QT-REFL})$$

$$\frac{\Gamma \vdash T_1 \equiv T_2 :: K}{\Gamma \vdash T_2 \equiv T_1 :: K} \quad (\text{QT-SYM})$$

$$\frac{\Gamma \vdash T_1 \equiv T_2 :: K \quad \Gamma \vdash T_2 \equiv T_3 :: K}{\Gamma \vdash T_1 \equiv T_3 :: K} \quad (\text{QT-TRANS})$$

System λLF : Term Equivalence

$$\frac{\Gamma \vdash S_1 \equiv S_2 :: * \quad \Gamma, x:S_1 \vdash t_1 \equiv t_2 : T}{\Gamma \vdash \lambda x:S_1. t_1 \equiv \lambda x:S_2. t_2 : (x:S_1) \rightarrow T} \quad (\text{Q-ABS})$$

$$\frac{\Gamma \vdash t_1 \equiv s_1 : (x:S) \rightarrow T \quad \Gamma \vdash t_2 \equiv s_2 : S}{\Gamma \vdash t_1 t_2 \equiv s_1 s_2 : [x \mapsto t_2]T} \quad (\text{Q-APP})$$

$$\frac{\Gamma, x:S \vdash t : T \quad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x:S. t) s \equiv [x \mapsto s]t : [x \mapsto s]T} \quad (\text{Q-BETA})$$

$$\frac{\Gamma \vdash t : (x:S) \rightarrow T \quad x \notin FV(t)}{\Gamma \vdash \lambda x:S. t x \equiv t : (x:S) \rightarrow T} \quad (\text{Q-ETA})$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \equiv t :: T} \quad (\text{Q-REFL})$$

$$\frac{\Gamma \vdash t \equiv s : T}{\Gamma \vdash s \equiv t : T} \quad (\text{Q-SYM})$$

$$\frac{\Gamma \vdash t_1 \equiv t_2 : T \quad \Gamma \vdash t_2 \equiv t_3 : T}{\Gamma \vdash t_1 \equiv t_3 : T} \quad (\text{Q-TRANS})$$

Strong Normalization

Given the following β -reduction rules

$$\frac{t_1 \longrightarrow t'_1}{\lambda x: T_1. t_1 \longrightarrow \lambda x: T_1. t'_1} \quad (\beta\text{-ABS})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \quad (\beta\text{-APP1})$$

$$\frac{t_2 \longrightarrow t'_2}{t_1 t_2 \longrightarrow t_1 t'_2} \quad (\beta\text{-APP2})$$

$$(\lambda x: T_1. t_1) t_2 \longrightarrow [x \mapsto t_2] t_1 \quad (\beta\text{-APPAbs})$$

Theorem [Strong Normalization]: if $\Gamma \vdash t : T$, then there is no infinite sequence of terms t_i such that $t = t_1$ and $t_i \longrightarrow t_{i+1}$.

The Calculus of Constructions

The Calculus of Constructions: Syntax

$t ::=$

s

x

$\lambda x:t.t$

$t t$

$(x:t) \rightarrow t$

terms

sort

variable

abstraction

application

dependent type

$s ::=$

$*$

\square

sorts

sort of proper types

sort of kinds

$\Gamma ::=$

\emptyset

$\Gamma, x:T$

contexts

empty context

term variable binding

The semantics is the usual β -reduction.

The Calculus of Constructions: Typing

$$\vdash * : \square \text{ (T-AXIOM)} \qquad \frac{x:T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$$

$$\frac{\Gamma \vdash S : s_1 \quad \Gamma, x:S \vdash t : T}{\Gamma \vdash \lambda x:S.t : (x:S) \rightarrow T} \text{ (T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : (x:S) \rightarrow T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1 t_2 : [x \mapsto t_2]T} \text{ (T-APP)}$$

$$\frac{\Gamma \vdash S : s_1 \quad \Gamma, x:S \vdash T : s_2}{\Gamma \vdash (x:S) \rightarrow T : s_2} \text{ (T-PI)}$$

$$\frac{\Gamma \vdash t : T \quad T \equiv T' \quad \Gamma \vdash T' : s}{\Gamma \vdash t : T'} \text{ (T-CONV)}$$

The equivalence relation $T \equiv T'$ is based on β -reduction.

Four Kinds of Lambdas

| Example | Type |
|--|--|
| $\lambda x:\mathbb{N}.x + 1$ | $\mathbb{N} \rightarrow \mathbb{N}$ |
| $\lambda f:\mathbb{N} \rightarrow \mathbb{N}.f\ x$ | $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ |

Four Kinds of Lambdas

| Example | Type |
|---|---|
| $\lambda x:\mathbb{N}.x + 1$ | $\mathbb{N} \rightarrow \mathbb{N}$ |
| $\lambda f:\mathbb{N} \rightarrow \mathbb{N}.f\ x$ | $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ |
| $\lambda X:*. \lambda x:X. x$ | $(X:*) \rightarrow X \rightarrow X$ |
| $\lambda F:* \rightarrow *. \lambda x:F\ \mathbb{N}. x$ | $(F:* \rightarrow *) \rightarrow (F\ \mathbb{N}) \rightarrow (F\ \mathbb{N})$ |

Four Kinds of Lambdas

| Example | Type |
|---|---|
| $\lambda x:\mathbb{N}.x + 1$ | $\mathbb{N} \rightarrow \mathbb{N}$ |
| $\lambda f:\mathbb{N} \rightarrow \mathbb{N}.f\ x$ | $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ |
| $\lambda X:*. \lambda x:X. x$ | $(X:*) \rightarrow X \rightarrow X$ |
| $\lambda F:* \rightarrow *. \lambda x:F\ \mathbb{N}. x$ | $(F:* \rightarrow *) \rightarrow (F\ \mathbb{N}) \rightarrow (F\ \mathbb{N})$ |
| $\lambda X:*. X$ | $* \rightarrow *$ |
| $\lambda F:* \rightarrow *. F\ \mathbb{N}$ | $(* \rightarrow *) \rightarrow *$ |

Four Kinds of Lambdas

| Example | Type |
|---|---|
| $\lambda x:\mathbb{N}.x + 1$ | $\mathbb{N} \rightarrow \mathbb{N}$ |
| $\lambda f:\mathbb{N} \rightarrow \mathbb{N}.f\ x$ | $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ |
| $\lambda X:*. \lambda x:X. x$ | $(X:*) \rightarrow X \rightarrow X$ |
| $\lambda F:* \rightarrow *. \lambda x:F\ \mathbb{N}. x$ | $(F:* \rightarrow *) \rightarrow (F\ \mathbb{N}) \rightarrow (F\ \mathbb{N})$ |
| $\lambda X:*. X$ | $* \rightarrow *$ |
| $\lambda F:* \rightarrow *. F\ \mathbb{N}$ | $(* \rightarrow *) \rightarrow *$ |
| $\lambda n:\mathbb{N}. \text{Vec } n$ | $\mathbb{N} \rightarrow *$ |
| $\lambda f:\mathbb{N} \rightarrow \mathbb{N}. \text{Vec } (f\ 6)$ | $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow *$ |

Strong Normalization

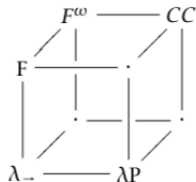
Theorem [Strong Normalization]: if $\Gamma \vdash t : T$, then there is no infinite sequence of terms t_i such that $t = t_1$ and $t_i \longrightarrow t_{i+1}$.

Question : Why the property is important?

Pure Type Systems

$$\frac{\Gamma \vdash S : s_i \quad \Gamma, x:S \vdash T : s_j}{\Gamma \vdash (x:S) \rightarrow T : s_j} \quad (\text{T-PI})$$

| System | (s_i, s_j) |
|-------------------------|--|
| λ_{\rightarrow} | { (*, *) } |
| λP | { (*, *), (*, \square) } |
| F | { (*, *), (\square , *) } |
| F^{ω} | { (*, *), (\square , *), (\square , \square) } |
| CC | { (*, *), (*, \square), (\square , *), (\square , \square) } |



The system λP is $\lambda L F$ in PTS-style.

Dependent Types in Practice

Proof Assistants

Dependent type theories are at the foundation of proof assistants, like Coq, Agda, etc.

By *Curry-Howard Correspondence*

- ▶ proofs \longleftrightarrow programs
- ▶ propositions \longleftrightarrow types

Coq is based on *Calculus of Inductive Construction*, which is an extension of CC with inductive definition.

Proofs in Coq: Example

```
Inductive nat : Type :=  
  | 0  
  | S (n : nat).
```

```
Fixpoint double (n : nat) : nat :=  
  match n with  
  | 0 => 0  
  | S n' => S (S (double n'))  
end.
```

```
Inductive even : nat -> Prop :=  
  | even0 : even 0  
  | evenS : forall x:nat, even x -> even (S (S x)).
```

Proofs in Coq: Example, Continued

```
Definition even_prop := forall x:nat, even (double x).
```

```
Fixpoint even_rec(m: nat)(p0: (even (double 0)))  
(pS: forall n:nat,  
      (even (double n)) -> (even (double (S n))))  
: even (double m) :=  
  match m with  
  | 0 => p0  
  | S n' => pS n' (even_rec n' p0 pS)  
end.
```

```
Definition even_proof: even_prop :=  
  fun n => even_rec n even0  
    (fun m evenN => (evenS (double m) evenN)).
```

Dependent Types in Programming Languages

Despite the huge success in proof assistants, its adoption in programming languages is limited.

- ▶ Scala supports *path-dependent types* and *literal types*.
- ▶ Dependent Haskell is proposed by researchers.

Dependent Types in Programming Languages

Despite the huge success in proof assistants, its adoption in programming languages is limited.

- ▶ Scala supports *path-dependent types* and *literal types*.
- ▶ Dependent Haskell is proposed by researchers.

Challenge: the decidability of type checking.

Problem with Type Checking: Vector Again

Value constructors:

$$\begin{aligned} \text{Vec} & : \mathbb{N} \rightarrow * \\ \text{nil} & : \text{Vec } 0 \\ \text{cons} & = \lambda n:\mathbb{N}. D \rightarrow \text{Vec } n \rightarrow \text{Vec } n + 1 \end{aligned}$$

Appending vectors:

$$\begin{aligned} \text{append} & : (m:\mathbb{N}) \rightarrow (n:\mathbb{N}) \rightarrow \text{Vec } m \rightarrow \text{Vec } n \rightarrow \text{Vec } (m + n) \\ \text{append} & = \lambda m:\mathbb{N}. \lambda n:\mathbb{N}. \lambda l:\text{Vec } m. \lambda t:\text{Vec } n. \\ & \quad \text{match } l \text{ with} \\ & \quad | \text{nil} \Rightarrow t \\ & \quad | \text{cons } r \times y \Rightarrow \text{cons } (r + n) \times (\text{append } r \ n \ y \ t) \end{aligned}$$

Question: How does the type checker know $r + 1 + n = r + n + 1$?