Well-typed programs do not 'go wrong'

Type Systems, Type Inferencing

LECTURE 6

With some slides from Andres Löh

TYPE SYSTEMS

A type-correct program only applies functions to values it is defined for!

"Well-typed programs do not 'go wrong'"

-- Robin Milner

Catch hard-to-find typo's and logical errors at compile time!

No runtime checks for verifying type correctness needed.

Type-Inferencing

The type-inferencer in a compiler…

- … checks, whether a program is correctly typed according to the type rules of the language.
- … automatically determines the type of every expression in a program.

Damas-Hindley-Milner Type Inferencer

In this lecture we will look at the Damas-Hindley-Milner type system.

- Better known as Damas-Milner or Hindley-Milner type system.

Damas-Hindley-Milner Type Inferencer

The system has been invented by Roger Hindley (1969).

Rediscovered by Robin Milner (1978).

Luis Damas contributed formal analyses and proofs (1982).

Damas-Hindley-Milner Type Inferencer

The type system was originally used in ML.

Today many of the ML-family languages use this system, and Haskell also uses a variant of it!

Damas-Hindley-Milner Type Inferencer

In the paper “Principal type-schemes for functional programs” by Damas & Milner (1982) there is a clear separation between to aspects:

1. The type system. A deductive proof system that allows multiple outcomes, and multiple ways to an outcome. (First half of this lecture)

2. The inference algorithm (Algorithm *W*). Step-by-step procedure, no choices, one outcome! (Second half of this lecture).

The Key Idea

The Damas-Milner algorithm distinguishes lambda-bound and let-bound (term) variables:

- Lambda-bound variables are always assumed to have a monotype;
- Of let-bound variables, we know what they are bound to, therefore they can have polymorphic types.

Inference Variables

Whenever a lambda-bound variable is encountered, a fresh inference variable is introduced.

The variable represents a monotype!

When we learn more about the types, inference variables can be substituted by types.

Inference variables are different from universally quantified variables that express polymorphism!

A Small Language

```
e ::= c                          -- constants
    | x                          -- variables
    | e e'                       -- application
    | λx -> e                    -- abstraction
    | let x = e in  e'  -- let binding
    | if e then e' else e''
```

Some Primitive Functions

```
null :: [a] → Bool
nil  :: [a]
cons :: a → [a] → [a]
hd   :: [a] → a
tl   :: [a] → [a]
```

Polymorphism

$$id :: a \rightarrow a$$

What we really mean to say is:

$$id :: \forall a.\ a \rightarrow a$$

In Haskell all type variables are implicitly universally quantified.

Today we do make a distinction between the two.
Unbound variables stand for one type!

The Type Language

τ ::= a | τ_c | [τ] | $\tau \rightarrow \tau'$
τ_c ::= Int | Bool

Monotypes

σ ::= τ | $\forall$a.σ

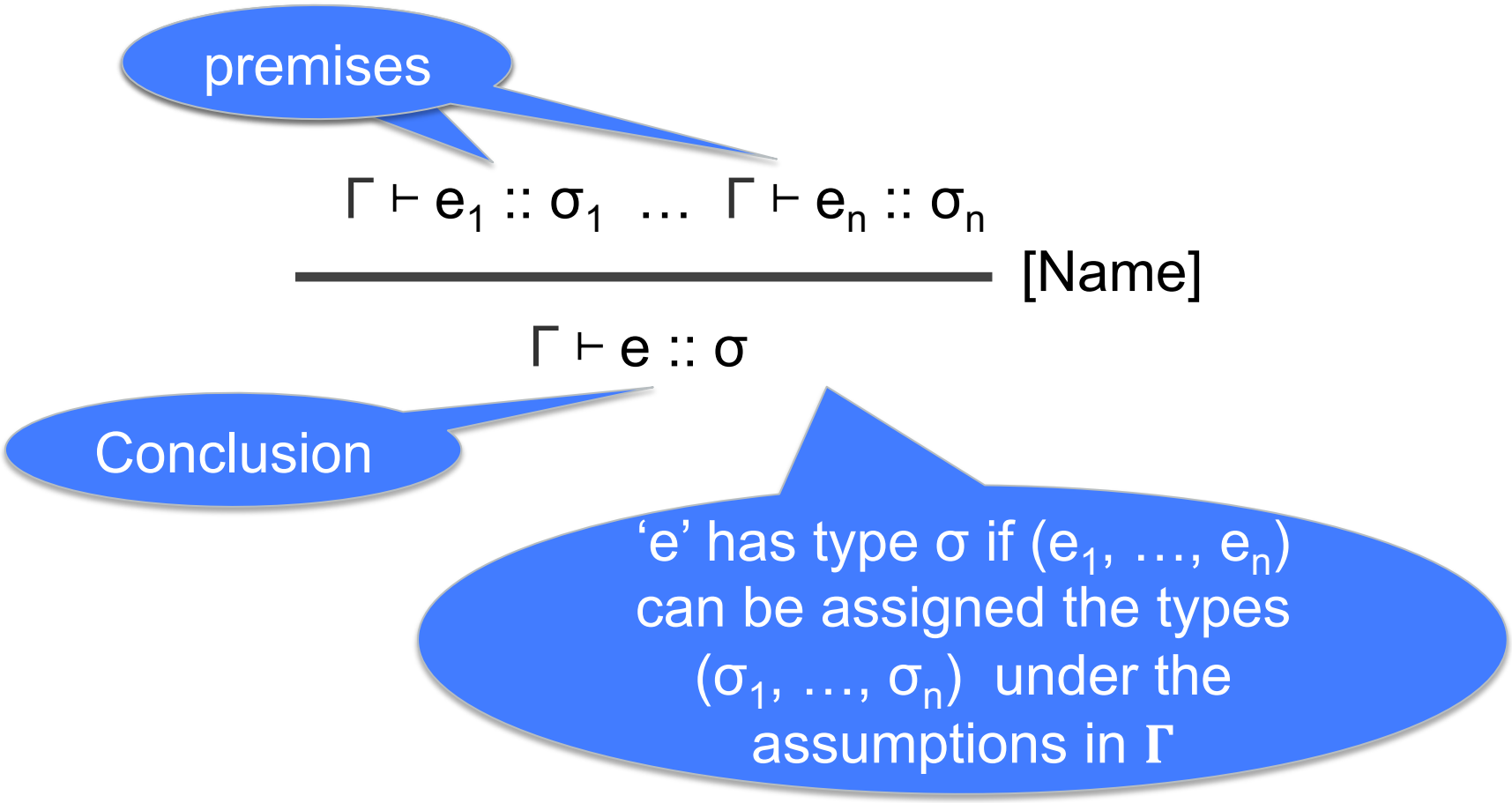
Polytypes

Γ ::= Γ,x $\mapsto$ σ | ε

If x occurs twice or in the type environment the right most x shadows all others

Derivation Trees

An Inference Rule



Constants

In general:

$$\frac{}{\Gamma \vdash c :: \tau_c}\ [\text{Const}]$$

Or more specifically:

$$\frac{i \in \text{Int}}{\Gamma \vdash c :: \tau_c}\ [\text{Int}] \qquad \frac{b \in \{\text{True}, \text{False}\}}{\Gamma \vdash b :: \text{Bool}}\ [\text{Bool}]$$

Variables

$$\frac{(x \mapsto \sigma) \in \Gamma}{\Gamma \vdash x :: \sigma}\ [\text{Var}]$$

Arithmetic Operators

$$\frac{\Gamma \vdash e_1 :: \mathrm{Int} \qquad \Gamma \vdash e_2 :: \mathrm{Int}}{\Gamma \vdash e_1 \oplus e_2 :: \mathrm{Int}} \text{ [Arith-Op]}$$

where $\oplus \in \{+,-.*,/\}$

Conditionals

$$\frac{\Gamma \vdash e_1 :: \textbf{Bool} \quad \Gamma \vdash e_2 :: \tau \quad \Gamma \vdash e_3 :: \tau}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 :: \tau} \text{ [If]}$$

Let

$$\frac{\Gamma \vdash e_1 :: \sigma \qquad \Gamma, x \mapsto \sigma \vdash e_2 :: \tau}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 :: \tau} \text{ [Let]}$$

Abstraction

$$\frac{\Gamma, x \mapsto \tau_x \vdash e :: \tau}{\Gamma \vdash \lambda x \to e :: \tau_x \to \tau}\ [\text{Abs}]$$

Application

$$\frac{\Gamma \vdash e_1 :: \tau_x \rightarrow \tau \qquad \Gamma \vdash e_2 :: \tau_x}{\Gamma \vdash e_1\ e_2 :: \tau}\ [App]$$

Example

$(x \mapsto \text{Int}) \in \varepsilon, x \mapsto \text{Int}$ [Var]

$\varepsilon, x \mapsto \text{Int} \vdash \mathbf{x :: Int}$

$\varepsilon, x \to \text{Int} \vdash \mathbf{1 :: Int}$ [Const]

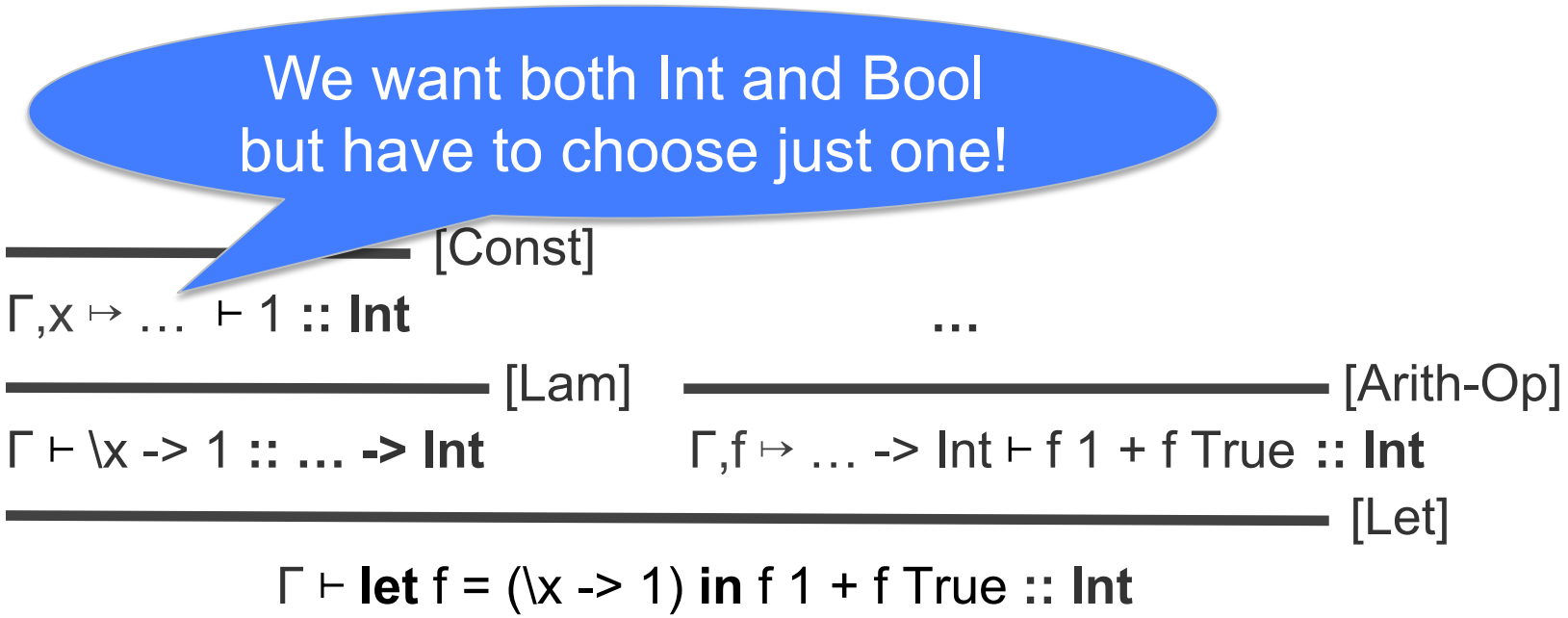
$\varepsilon, x \mapsto \text{Int} \vdash \mathbf{x + 1 :: Int}$ [Arith-Op]

$\varepsilon \vdash \mathbf{(\backslash x \to x + 1) :: Int \to Int}$ [Abs]

$\varepsilon \vdash \mathbf{41 :: Int}$ [Const]

$\varepsilon \vdash \mathbf{(\backslash x \to x + 1)\ 41 :: Int}$ [App]

Example



Generalisation

$$\frac{\Gamma \vdash e :: \sigma \quad a \notin ftv(\Gamma)}{\Gamma \vdash e :: \forall a.\sigma} \text{ [Gen]}$$

The function “ftv” returns the set of all unbound variables in a type environment.

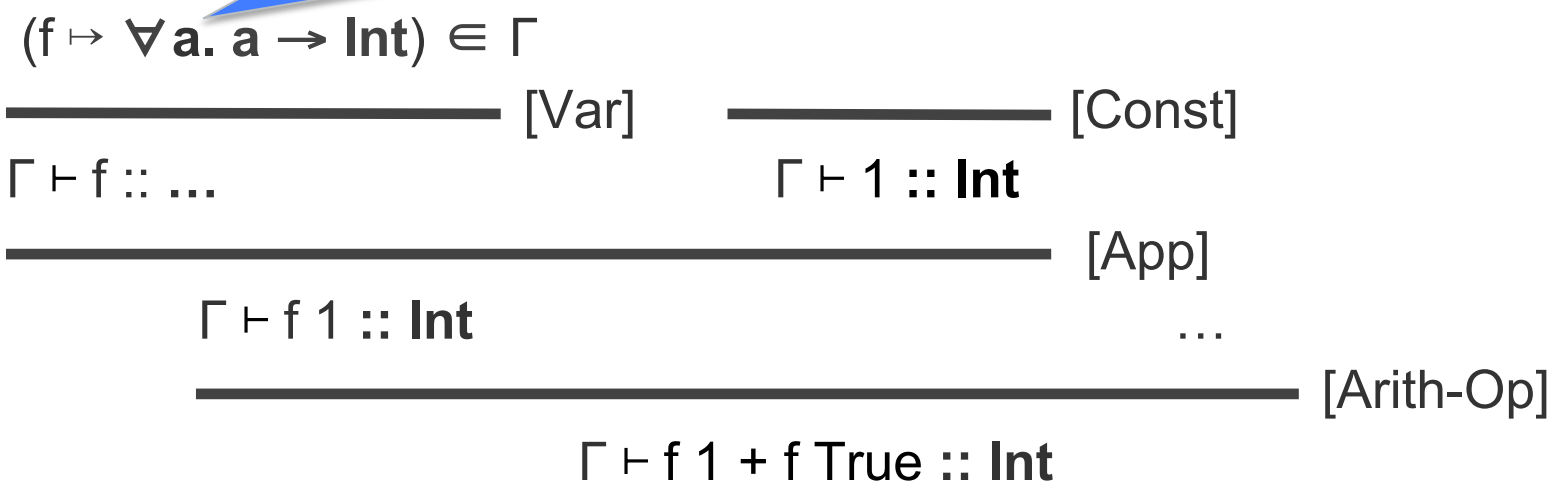
Example

$$\dfrac{\dfrac{\dfrac{}{\Gamma,x \mapsto \mathbf{a} \vdash 1 :: \mathbf{Int}}\,[\text{Const}]}{\Gamma \vdash \backslash x \text{ -> } 1 :: \mathbf{a \text{ -> } Int}}\,[\text{Lam}]}{\Gamma \vdash \backslash x \text{ -> } 1 :: \forall \mathbf{a.\ a \text{ -> } Int}}\,[\text{Gen}] \qquad \dfrac{\dots}{\Gamma,f \mapsto \forall \mathbf{a.\ a \text{ -> } Int} \vdash f\ 1 + f\ \text{True} :: \mathbf{Int}}\,[\text{Arith-Op}]$$

$$\dfrac{\text{above}}{\Gamma \vdash \mathbf{let}\ f = (\backslash x \text{ -> } 1)\ \mathbf{in}\ f\ 1 + f\ \text{True} :: \mathbf{Bool}}\,[\text{Let}]$$

Example

We have to somehow lose the quantification!



Substitutions

A substitution is usually denoted as: $[\tau_1/a_1, \ldots, \tau_n/a_n]$.

A substitution replaces a_i by τ_i if a_i is a free variable in a type scheme σ.

Example:

$[b/a_1, Int/a_2, Int/a_3]\ (\forall a_2.\ a_1 \rightarrow a_2 \rightarrow a_3)$
$\Rightarrow (\forall a_2.\ b \rightarrow a_2 \rightarrow Int)$

Instances

In contrast to substitutions instantiating types works on bound type variables.

A type scheme σ' is an instance of σ if:

- All bound variables of σ' are also bound in σ.
- Some bound variables of σ are substituted in σ'

If σ' is an instance of σ we write: $\sigma > \sigma'$

Instances

$\forall a_1\ a_2\ a_3.\ a_1 \to a_2 \to a_3 \ > \ \forall a_2\ a_3.\ b \to a_2 \to a_3$ 

$\forall a_1\ a_2\ a_3.\ a_1 \to a_2 \to a_3 \ > \ \forall a_2.\ b \to a_2 \to \text{Int}$ 

$\forall a_3.\ a_1 \to \text{Int} \to a_3 \ > \ \forall a_1\ a_3.\ a_1 \to \text{Int} \to a_3$ 

Instantiation

$$\frac{\Gamma \vdash e :: \sigma \quad \sigma > \sigma'}{\Gamma \vdash e :: \sigma'} \text{ [Inst]}$$

Example

$$\frac{(f \mapsto \forall a.\ a \to \mathbf{Int}) \in \Gamma}{\Gamma \vdash f :: \forall a.\ a \to \mathbf{Int}}\ [\text{Var}]$$

$$\frac{\Gamma \vdash f :: \mathbf{Int} \to \mathbf{Int} \qquad \Gamma \vdash 1 :: \mathbf{Int}}{\Gamma \vdash f\ 1 :: \mathbf{Int}}\ [\text{App}] \qquad \frac{}{\Gamma \vdash 1 :: \mathbf{Int}}\ [\text{Const}]$$

$$\frac{\Gamma \vdash f\ 1 :: \mathbf{Int} \qquad \ldots}{\Gamma \vdash f\ 1 + f\ \text{True} :: \mathbf{Int}}\ [\text{Arith-Op}]$$

The Type System

The type system presented can be used to assign any valid type to an expression, not just a principal type scheme.

For example:

$$\frac{(x \mapsto \mathbf{Int}) \in \varepsilon, x \mapsto \mathrm{Int}}{\varepsilon, x \mapsto \mathbf{Int} \vdash x :: \mathbf{Int}}\ [\mathrm{Var}]$$

$$\frac{\varepsilon, x \mapsto \mathbf{Int} \vdash x :: \mathbf{Int}}{\varepsilon \vdash (\backslash x \rightarrow x) :: \mathbf{Int} \rightarrow \mathbf{Int}}\ [\mathrm{Abs}]$$

TYPE INFERENCING

The Type System

Can be used to verify whether a certain type can be assigned to an expression.

It doesn't necessarily find the principal type.

It also is not syntax directed.

- The *inst* and *gen* rules can be used everywhere.

Relies on making correct assumptions (predicting what we will need later on).

A Type Inferencing Algorithm

A Type Inferencing Algorithm should always find the principal type without any further input and reject ill-typed programs.

Syntax Directed Type System

The type system we defined is not syntax directed, we can always use the instantiation and generalisation rules.

In order to use the type system as the basis for an algorithm we have to decide exactly where we wish to use these rules!

- Preferably without rejecting more programs!

Combining Inst and Var Rules

In the type environment we store type schemes but our type rules deal mainly with types.

It seems to be a good fit to instantiate a type the moment we get it from the type environment!

Combing Inst and Var Rules

We define a function *inst* that fully instantiates a type scheme.

```
inst :: TypeScheme → Type
inst (∀a1 … an. τ) = [a1'/a1, … an'/an] τ
  where a1', …, an' are fresh variables
```

A variable ‘a’ is called fresh if it has not been used before.

Combining Inst and Var Rules

$$\frac{(x \mapsto \sigma) \in \Gamma \quad \tau = \text{inst}(\sigma)}{\Gamma \vdash x :: \tau}\ [\text{Var}]$$

Combining Gen and Let Rules

In the type environment we store type schemes but our type rules give us just types.

When we have fully determined an expressions type we can generalise! In our language this is the case in let bindings.

Combining Gen and Let Rules

We define a function *gen* that generalises a type as far as possible.

- We cannot generalise over type variables in the environment

```
gen :: Env → Type → TypeScheme
gen Γ τ = ∀a1 … an. τ
  where {a1, …, an} = ftv(τ) – ftv(Γ)
```

Combining Gen and Let Rules

$$\frac{\Gamma \vdash e_1 :: \boldsymbol{\tau}_1 \qquad \sigma = \text{gen}\ \Gamma\ \boldsymbol{\tau}_1 \qquad \Gamma, x \mapsto \sigma \vdash e_2 :: \boldsymbol{\tau}_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: \boldsymbol{\tau}_2}\ [\text{If}]$$

Unification Algorithm

The Inference Algorithm needs a way to unify two types.

The unification of two types τ_1 and τ_2 results in a substitution S such that: S $\tau_1 \equiv$ S τ_2.

Two substitutions S_1 and S_2 can be combined into one substitution S, denoted by $S_1 \circ S_2$, such that:
S $\tau \equiv S_1(S_2\ \tau)$.

Unification Algorithm

```
u :: Type → Type → Subst
u Int        Int        = []
u Bool       Bool       = []
u [a]        [b]        = u a b
u (a → b) (c → d) = let s1 = u a c
                        s2 = u (s1 b) (s1 d)
                    in s2∘s1
u τ          a          = if (τ ≡ a || a ∉ ftv(τ))
                          then [τ/a] else fail
u a          τ          = if (τ ≡ a || a ∉ ftv(τ))
                          then [τ/a] else fail
u _          _          = fail
```

Algorithm *W*

One algorithm to compute the principal type of an expression is Algorithm *W*.

```
w :: Env → Expr → (Type, Subst)
```

Algorithm *W*

$$\frac{i \in \text{Int}}{\Gamma \vdash c :: \tau_c}\,[\text{Int}] \qquad \frac{b \in \{\text{True}, \text{False}\}}{\Gamma \vdash b :: \text{Bool}}\,[\text{Bool}]$$

```
w :: Env → Expr → (Type, Subst)
w _  i = (Int,  [])
w _  b = (Bool, [])
```

Algorithm *W*

$$\frac{(x \mapsto \sigma) \in \Gamma \quad \tau = \text{inst}(\sigma)}{\Gamma \vdash x :: \tau}\ [\text{Var}]$$

```
w :: Env → Expr → (Type, Subst)
w Γ  x = case Γ  x of
           Just σ → (inst σ, [])
           Nothing → fail
```

Algorithm *W*

$$\frac{\Gamma \vdash e_1 :: \mathsf{Int} \qquad \Gamma \vdash e_2 :: \mathsf{Int}}{\Gamma \vdash e_1 \oplus e_2 :: \mathsf{Int}}\ [\text{Arith-Op}]$$

```
w :: Env → Expr → (Type, Subst)
w Γ  (e1 ⊕ e2) = let (τ1, S1) = w Γ  e1
                     (τ2, S2) = w (S1 Γ) e2
                     S3 = u (S2 τ1) Int
                     S4 = u (S3 τ2) Int
                 in (Int, S4∘S3∘S2∘S1)
```

Algorithm *W*

$$\frac{\Gamma \vdash e_1 :: \mathrm{Bool} \qquad \Gamma \vdash e_2 :: \tau \qquad \Gamma \vdash e_3 :: \tau}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 :: \tau} \text{ [If]}$$

```
w :: Env → Expr → (Type, Subst)
w Γ  (if e1 then e2 else e3) =
        let (τ1, S1) = w Γ  e1
            (τ2, S2) = w (S1 Γ) e2
            (τ3, S3) = w (S2∘S1 Γ) e3
            S4 = u (S3∘S2 τ1) Bool
            S5 = u (S4 τ3) (S4∘S3 τ2)
        in (S5∘S4 τ3, S5∘S4∘S3∘S2∘S1)
```

Algorithm *W*

$$\frac{\Gamma \vdash e_1 :: \tau_1 \quad \sigma = \text{gen}\ \Gamma\ \tau_1 \quad \Gamma, x \mapsto \sigma \vdash e_2 :: \tau_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: \tau_2}\ [\text{If}]$$

```
w :: Env → Expr → (Type, Subst)
w Γ  (let x = e1 in e2) =
        let (τ1, S1) = w Γ e1
            σ = gen Γ τ1
            (τ2, S2) = w (S1 Γ ,x ↦ σ) e2
        in (τ2, S2∘S1)
```

Algorithm *W*

$$\frac{\Gamma, x \mapsto \tau_x \vdash e :: \tau}{\Gamma \vdash \lambda x \to e :: \tau_x \to \tau}\ [\text{Abs}]$$

```
w :: Env → Expr → (Type, Subst)
w Γ  (λx → e) =
        let a_x be fresh
            (τ, S_1) = w (Γ,x ↦ a_x) e_1
         in (S_1(a_x → τ), S_1)
```

Algorithm *W*

$$\frac{\Gamma \vdash e_1 :: \tau_x \to \tau \qquad \Gamma \vdash e_2 :: \tau_x}{\Gamma \vdash e_1\ e_2 :: \tau} \text{[App]}$$

```
w :: Env → Expr → (Type, Subst)
w Γ (e1 e2) =
      let ar be fresh
          (τ1, S1) = w Γ e1
          (τ2, S2) = w (S1 Γ) e2
          S3 = u (S2 τ1) (τ2 → ar)
      in (S3 ar, S3∘S2∘S1)
```

Reading material

- Principal type-schemes for functional programs
  - Luis Damas & Robin Milner

- The Implementation of Functional Programming Languages
  - Chapter 8 & 9
  - Book by Simon Peyton Jones