

(4.0)

Type checking

A type checker must find all errors (or almost all) in the specification

lets see:

sort: List → List

↑

This means that: sort([2, 3, -1]),
is type - correct

but sort(0); it's not.

However, there is nothing that prevents the definition of a sorting function that just returns the same list back.

In languages like Agda, you can use propositions as types principle, which in particular makes possible to express specification as types.

For instance:

Sort : (x : hist) \rightarrow (y : hist) &
Sorted(x, y)

Type checking has another function apart from correctness control, it is used for type annotations, this enables the compiler to produce more efficient machine code.

Specifying a type checker

9.2 There is

There is no specific-domain language to implement type checking, so we have to write them in a general purpose programming language.

But, there are standard notations that can be used for specifying the type system, and easily converted to code in any host language.

The most common is inference rules (this is like a deduction, conjecture)

for example:

$$\begin{array}{ll} \underline{a : \text{bool}} & \underline{b : \text{bool}} \\ a \& \& b : \text{bool} \end{array}$$

if a has type bool, and b has type bool

then, $a \& b$ has type bool.

In general, a inference rule has premises and a conclusion

$$\frac{\Gamma_1 \dots \Gamma_n}{\Gamma}$$

From the premissed $\Gamma_1 \dots \Gamma_n$, we can conclude Γ

In type checker, the rule is applied upside down:

To check J , check $J_1 \dots J_n$



The premisses and conclusions in inference rules are called judgements

$$\frac{\text{judgements}}{\text{judgment}} \rightarrow \frac{J_1 \dots J_n}{J}$$



The most common judgements in type systems have the form

$$e : T$$

↑ has
expression of type T

4.3 Type checking and Type inference

→ first step from inference rule
to implementation is pseudo code for
syntax-directed translation.



this will be treated as clauses in
a recursive function definition that
traverses expression trees. They are
read upside down, that is;



To check J_n ; check $J_1 \dots J_n$



which is converted to program code
of the format;



$J:$

J_1

\dots

J_n

There are Two kinds of functions



Type checking

given an e and a T , decide if $e : T$

Type inference

given an e , find T such that $e : T$.



When we translate a typing rule to type checking code, its conclusion becomes a case for pattern matching and its premises become recursive calls for type checking. For instance, the above & g rule becomes

Check($a \& b$, bool):

check(a , bool)

check(b , bool)

In a type inference rule, the premises become recursive calls as well, but the type in the conclusion becomes the value returned by the function:

```
infer(a & b):
    check(a, bool)
    check(b, bool)
    return bool
```

Here we use concrete syntax (both in pseudo-code and inference rules)

$(a \& b)$ rather than $(EAnd\ ab)$

In real type checking code, abstract syntax must be used.

(4.3) Context, environment, and side conditions

Variables symbols like x can have any of the types available in a programming language.

The type it has depends on the context. In C or Java the context is determined by declarations of variables.

It is a data structure where one can look up a variable and get its type.

So, we can think that context is a lookup table of (variable, type) pairs.

In inference rules, the context is denoted by Γ , Δ .

(I will use a B because it's easier)

So to include the context (our look up table with their variables and types) we will now denote the judgement form for typing to:

$B \vdash e : T$

↑
expression e has T type in
context B

For example, the following judgement holds:

$x : \text{int}, y : \text{int} \vdash x + y > y : \text{bool}$

It means that $x + y \geq g$ is a boolean expression in the context where x and y are integer variables.

So, the notation for contexts:



$x_1 : T_1, \dots, x_n : T_n$



This is handy when we want to add one more variable, because we only need to add a ' $x : T$ '



$B, x : T$



context + new variable

Most typing rules are generalized by adding the same Γ (tables of $x:T$) to all judgments, because the context does not change.

$$\frac{\Gamma \vdash a : \text{bool} \quad \Gamma \vdash b : \text{bool}}{\Gamma \vdash a \& b : \text{bool}}$$

of course, this is not always the case ↓

The context does change in the rules for type checking derivations.

↓

The places where we need context, are those that involve variables

The
Typing rule for variable
expression is

↓

$\frac{}{B \vdash x : T}$ if $x : T$ in B

↗

This is not a judgement, is
a side condition, because
is a sentence line in English.

↓

This becomes clear if we
look at the pseudo-code!

↓

infer(B, x):
 $t := \text{lookup}(x, B)$
return t

looking up the type of the variable (Γ ; the context, which we remember, is a data structure like a table of $(\text{var}, \text{type})$) is not a recursive call to infer or check, but uses another function, $\text{lookup}(\cdot)$.

↓
lets see this in actual implementation code in Haskell

$\text{inference} \quad \left\{ \begin{array}{l} \text{infer}(\Gamma, x) \\ \text{pseudo code} \end{array} \right. \quad \left\{ \begin{array}{l} t := \text{lookup}(x, \Gamma) \\ \text{return } t \end{array} \right.$

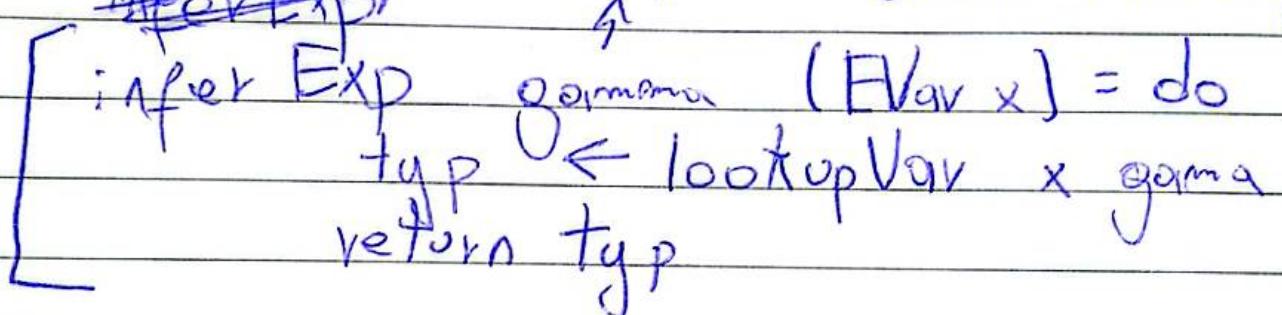
$\text{astell} \quad \left\{ \begin{array}{l} \text{inferExp} : \text{Context} \rightarrow \text{Exp} \rightarrow \text{EnvType} \\ \text{lookupVar} : \text{Ident} \rightarrow \text{Context} \rightarrow \text{EnvType} \end{array} \right.$

We also have to make the abstract syntax constructors explicit, we can't write just x , but $EVar x$, when we infer the type of a variable expression.



Γ

β



If the language has function definition, we also need to lookup the types of function when type checking function calls ($f(a, b, c)$),

So we will assume that the context β also include the types information for functions

so, a more appropriated name
for 'fc context' will be "environment"
(or jwt env) for type checking,
and not just the context.



The only place where the
function storage part of the env
changes, is when type checking
function's definition.

The only place where we
need it is when type checking
function calls.

The typing rules involve a look
up of the function in \mathcal{B} as aside
condition, and the typing arguments
as premises:

$\frac{B \vdash a_1 : T_1 \dots B \vdash a_n : T_n}{B \vdash f(a_1, \dots, a_n) : T}$ if $f : (T_1, \dots, T_n) \rightarrow T$
in B

$B \rightarrow$ environment in this case, not just context (because involves function types)

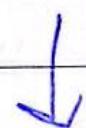
(4.5) Proofs in a type system

Inference rules are designed for the construction of proofs, which are structured as proof trees.

A proof tree can be seen as a trace of steps that the type checker performs when checking or inferring a type.

lets make a prof tree for the judgement $\rightarrow x: \text{int}, y: \text{int} \vdash x+y \geq y: \text{bool}$

↓
shown in the previous section



. Too large \rightarrow check page 62

In addition to variable rules,
we also use rules for " $+$ " and " \geq "



$B \vdash a : \text{int}$ $B \vdash b : \text{int}$

$\underline{B \vdash a+b : \text{int}}$

$\underline{B \vdash a : \text{int} \quad B \vdash b : \text{int}}$

$\underline{B \vdash a \geq b : \text{bool}}$



9.6 Overloading and Type conversions

Variables are examples of expression that can have different types in diff contexts (Rs)

But, another example is Overloaded operators. The binary arithmetic operators (+, -, *, /) and comparisons (==, !=, <, >, <=, >=) are in many languages usable for different types.

lets assume that the possible types for addition and comparisons are int, double, and string. The typing rules then look as follows:



Bt a:t Bf b:t if t:int, double or
Bt a+b:t String

B+ a : t B + b : t if t : int
double or string

B + a == b : bool

Notice that a "+" expression has the same type as its operands.

We can infer the type of the first and then check that the second has the same type.
Now the same

f

infer(a+b) :

t := infer(a)

// check that + ∈ (int, double, string)

check(b, t)

return t'

lets think about type conversions,
the general idea of type conversions
involves an ordering between types.
An object from a smaller type can
be safely (without loss of information)
converted to a larger type



So, the typing rule for addition
becomes



$$\frac{\text{B} \vdash a : t \quad \text{B} \vdash b : u}{\text{B} \vdash a + b : \text{max}(t, u)}$$

if $t, u \in \{\text{int, double, string}\}$

We assume that $\text{int} < \text{double} < \text{string}$

Exercise (4-0)

"3'hello £2"

(4.7) The validity of statements and function definitions

Expressing how types, which can be checked and inferred.

When we type check a statement, we are interested in whether it's valid or not.

For a validity statement, we need a new judgement form:

$B \vdash s \text{ valid}$

Statement s is valid in environment B

For instance, in a "while" statement the condition expression has to be boolean

$B \vdash e : \text{bool} . B \vdash s \text{ valid}$

$B \vdash \text{while}(e) . s \text{ valid}$

What about expressions used as statements, for instance, assignments and some function calls?

In that case, we don't need to care about what type of the expression is, just that it has one



The expression statement rule is



$\frac{B \vdash e : t}{B \vdash e; \text{ valid}}$

Similarly to statements, function definitions are just checked for validity



$\underline{x_1:T_1, x_m:T_m \vdash s_1 \dots s_n \text{ valid}}$

$T_f(T_1 x_1, T_m x_m) \{ s_1 \dots s_n \} \text{ valid}$

FIVISA

9.8 Declarations and block structures

Variables get their types in declarations. Each declaration has a scope scope, which is within a certain block. Blocks in C and Java correspond roughly to parts of code between curly brackets, {} and ;.

principles that regulate use of variables

1) A variable declared in a block, has its scope till the end of that block.

2) A variable can be declared again in an inner block, but not otherwise.

Examples in page 66

This means that a simple lookup table it's not enough, we need a stack of lookup tables, so R must be made into a stack of lookup tables.

We denote this with a dot notation, for example:

$B_1.B_2$

Where B_1 is an "old" (i.e. outer) context and B_2 an inner context. The innermost context is the top of the stack.

With a stack of contexts it starts by looking in the top most context and goes deeper in the stack only if it doesn't find the variable.

So, a declaration add a new variable to the current scope, which we check that it's fresh with respect to the context

to write a rule that check that a sequence of statements ~~are~~ ^{is} valid

→ $B \vdash s_1 \dots s_n \text{ valid}$

→ $\frac{\overline{B, x:T \vdash s_1 \dots s_n \text{ valid}}}{B \vdash x; s_1 \dots s_n \text{ valid}}$ $x \text{ not in } \text{the top-most context in } B$

Exercise 4-1

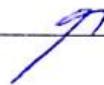
build a proof tree for the judgement

$\vdash \text{int } x; \quad x = x + 1; \text{ valid}$



$\frac{\beta \vdash x = \underbrace{x + 1}_{?}; \text{ valid}}{\beta \vdash x : \text{int}}$

$\beta \vdash x + 1 : \text{int} \quad (\text{add})$



$\beta \vdash x : \text{int}$
(var)

$\beta \vdash 1 : \text{int}$
(IntConst)

Exercise ⑨-2

`for (int i = 1; i < 10; i++) print(i);`

1- $\beta \vdash i : \text{int}$

$\frac{\beta \vdash e : \text{bool} \quad \beta \vdash \text{true}}{\beta \vdash \text{for}}$