4.0 Type checking

A type checker must find all errors (or almost alls) 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 sorted list back

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

for instance:

Sort : (x : list) → (y : list) & Sorted(x, y)

Type checking has another function apart from correctnes control, (it is used for type anotations), this enables the compiler to produce more efficat 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 ~~pour~~ purpose Programing language

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

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

for example:

$$\frac{a : \text{bool} \qquad b : \text{bool}}{a \,\&\&\, b : \text{bool}}$$

↑

if a has type bool, and b has type bool

then, a && b has type bool.

In general, a inference rule has premisses and a conclusion

$$\frac{J_1 \quad \dots \quad J_n}{J}$$

From the premisses J_1 to J_n, we can conclude J

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

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

↑

The most ~~comuit~~ 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 Sintax-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; check J1 .. Jn

↓

which is converted to program code of the format:

↓

J:
  J1
  ...
  Jn

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 become a case for pattern matching and its premisses become recursive calls for type checking. For instance, the above && rule become

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

FIVISA

---/---/---

In a type inference rule, the premisses 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 sintax (both in pseudo-code and inference rules)

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

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

4.5 Context, environment, and side conditions

Variables symbols like x can have any of the type avaible in a programing langage!

The type it has depends on the context. In C or Java the, 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, Gama.

(; will use a B because ;ts easyer)

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

$$B \vdash e : T$$

expression e has T type in contex B

For example, the following judgement holds:

$$x : int, y : int \vdash x + y > y : bool$$

It means that $x + y > 9$ is a boolean expression in the context where x and y are integer variables.

↓

So, the notation for contexts is:

↓

$x_1 : T_1, \ldots x_n : T_n$

↓

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 B (table of $x:T$) to all judgements, because the context does not change.

$$\frac{B \vdash a : bool \quad B \vdash b : bool}{B \vdash a \,\&\&\, b : bool}$$

of course, this is not always the case

↓

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

↓

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

The

Typing rule for variable expression is

↓

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

↑

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 := lookup(x, B)
    return t
```

Looking up the types of the variable (in the context, which we remember, is a data structure like a table of (var, type)) is not a recursive call to infer or check, but uses another function, lookup();

↓

Lets see this in actual implementation code in Haskell

↓

ference seudocode

```
infer(B, x);
    t := lookup(x, B)
    return t
```

askell ode

```
inferExp :: Context → Exp → Err Type
lookupVar :: Ident → Context → Err Type
```

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
B, but of coure we can't use it in that way

~~inferExp~~

```
inferExp gamma (EVar x) = do
    typ ← lookupVar x gamma
    return typ
```

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

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

---/---/---

So, a more appropiated name for the context will be "environment" (or just 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 B as a side condition, and the typing arguments as premisses:

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

$B \to$ environment in t_2 is 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's 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 proof tree for the judgement → x:int, y:int ⊢ x+y > y : bool

shown in the previous section

↓

too large → check page 62

In addition to variable rules, we also use rule for "+" and ">"

↓

$$\frac{B \vdash a : int \quad B \vdash b : int}{B \vdash a + b : int}$$

$$\frac{B \vdash a : int \quad B \vdash b : int}{B \vdash a > b : bool}$$

4.6 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 comparison (==, !=, <, >, <=, >=) 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:

$$\frac{R \vdash a : t \quad R \vdash b : t}{R \vdash a + b : t} \quad \text{if } t \text{ is int, double or string}$$

---/---/---

↓

$\frac{B \vdash a : t \quad B \vdash b : t}{B \vdash a == b : bool}$ if t: int, double or string

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 ~~its the same~~ type.
has the same

↓

```
infer (a+b):
    t := infer(a)
    // check that + ∈ (int, double, string)
    check(b, t)
    return t
```

Lets think about type convertions, the general idea of type convertions 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

↓

$$\frac{B \vdash a : t \qquad B \vdash b : u}{B \vdash a + b : \max(t, u)} \quad \text{if } t, u \in \{int, double, string\}$$

We assume that int < double < string

Exercise (4-0)

"3"hello"2"

4.7 The validity of statments and function definitions

Expressions have types, which can be checked and inferred.

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

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

$$B \vdash s \text{ valid}$$

statement s is valid in environment B

For intance, in a "while" statement the condition expression has to be bodean

$$B \vdash e : \text{bool} \quad B \vdash s \text{ valid}$$
$$B \vdash \text{while } (e)\ s \text{ valid}$$

What about expressions used as statments, 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 statment rule is

↓

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e;\ valid}$$

Similarly to statments, function definitions are just checked for validity

↓

$$\frac{x_1 : T_1, \ldots, x_m : T_m \vdash s_1 \ldots s_n\ valid}{T\ f(T_1\ x_1, \ldots, T_m\ x_m)\ \{s_1 \ldots, s_n\}\ valid}$$

FIVISA

9.8 Declarations and block Structures

Variables get their types in declarations. Each declaration has a ~~scoop~~ 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 enought, we need a stack of lookup tables, so B 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 secuence of statements ~~is~~ are valid

→ $B \vdash s_1 \dots s_n$ valid

→ $\dfrac{B\ x:T \vdash s_2 \dots s_n \text{ valid}}{B \vdash T\ x;\ s_2 \dots s_n \text{ valid}}$ x not in the top-most context in B

Exercise 4-1

build a proof tree for the judgement

⊢ int x ; x = x + 1 ; valid

↓

$B \vdash x = \underbrace{x + 1 ; \text{valid}}$

↑

$B \vdash x : int$ ↗

$B \vdash x + 1 : int$ (add) ↖

↗ ↑

$B \vdash x : int$ (var)

$B \vdash 1 : int$ (IntConst)

Exercise 4-2

for (int i = 1; i < 101; i++) print(i);

1- ~~$B \vdash i : int$~~

$B \vdash e : bool$ $B \vdash stmt$

$B \vdash for$