CS4613 Lecture 8: Types

David Bremner

March 10, 2025

Type Checking

static types checked before program execution

benefits limited proofs of correctness

strategy recursively evaluate the type of an expression

A language with numbers and strings

p. 110

```
(define-type BinOp
  [plus]
  [++]) ;; string concat

(define-type Exp
  [binE (operator : BinOp)
        (left : Exp)
        (right : Exp)]
  [numE (value : Number)]
  [strE (value : String)])
```

Some sample expressions

exp1

```
(binE (plus) (numE 3) (numE 4))  ;; OK
(binE (++) (strE "3") (strE "4")) ;; OK
(binE (plus) (numE 3) (strE "4")) ;; not OK
```

Some sample expressions

```
(binE (plus) (numE 3) (numE 4))  ;; OK
(binE (++) (strE "3") (strE "4")) ;; OK
(binE (plus) (numE 3) (strE "4")) ;; not OK
```

1. None of these are problematic from the point of view of expressions; our "grammar" does not try to enforce typing, although it (in principle) could do a partial job

An evaluator

```
(define (interp expr)
  (type-case Exp expr
    [(numE n) (numV n)]
    [(strE s) (strV s)]
    [(binE o l r)
     (let ([l-val (interp l)]
           [r-val (interp r)])
       (type-case BinOp o
         [(++) (on-strings string-append l-val r-val)]
         [(plus) (on-nums + l-val r-val)]))])))
```

Dynamic (run time) type checking

```
(define (on-nums func l r)
  (cond
    [(and (numV? l) (numV? r))
     (numV (func (numV-value l) (numV-value r)))]
    [else  (error 'interp "expected 2 numbers")]))
```

So we detect the error, what is the problem?

exp2

```
(test (interp (binE (plus) (numE 3) (numE 4)))
        (numV 7))
(test (interp (binE (++) (strE "3") (strE "4")))
      (strV "34"))
(test/exn (interp (binE (plus) (numE 3) (strE "4")))
          "numbers")
```

└─So we detect the error, what is the problem?

1. Our intepreter is *dynamically typed*, so it detects the typing error without attempting an invalid operation, or crashing
2. Our implementation language has more checks than something like `C`, so we are unlikely to make have undefined behaviour
3. The answer is that we may not detect the error until after the software is in use for some time. This can be very inconvenient/expensive to fix

So we detect the error, what is the problem?

```
(test (interp (binE (plus) (numE 3) (numE 4)))
        (numV 7))
(test (interp (binE (++) (strE "3") (strE "4")))
      (strV "34"))
(test/exn (interp (binE (plus) (numE 3) (strE "4")))
          "numbers")
```

“Evaluating” types

p. 112

Each type abstracts over a set of values.

```
(define-type Type
  [numT]
  [strT])

(define (tc e)
  (type-case Exp e
    [(binE o l r)
     (type-case BinOp o
       [(plus) (expect2 (numT) (tc l) (tc r))]
       [(++) (expect2 (strT) (tc l) (tc r))])]
    [(numE v) (numT)]
    [(strE v) (strT)]))
```

"Evaluating" types

p. 112

Each type abstracts over a set of values.

```
(define-type Type
  [numT]
  [strT])

(define (tc e)
  (type-case Exp e
    [(binE o l r)
     (type-case BinOp o
       [(plus) (expect2 (numT) (tc l) (tc r))]
       [(++) (expect2 (strT) (tc l)
    [(numE v) (numT)]
    [(strE v) (strT)]))
```

p. 112

1. This has been refactored from the code in the book to better fit on slides and to highlight the comparison with dynamic type checking

Compare and contrast

Dynamic

```
(define (on-strings func l r)
  (cond
    [(and (strV? l) (strV? r))
     (strV (func (strV-value l) (strV-value r)))]
    [else  (error 'interp "expected 2 numbers")]))
```

Static

```
(define (expect2 type lt rt)
  (cond
    [(and (equal? type lt) (equal? type rt)) type]
    [else (type-case Type type
            [(numT) (error 'tc "needs 2 numbers")]
            [(strT) (error 'tc "needs 2 strings")])]))
```

Compare and contrast

Dynamic

```
(define (on-strings func l r)
  (cond
    [(and (strV? l) (strV? r))
     (strV (func (strV-value l) (strV-value r)))]
    [else  (error 'interp "expected 2 numbers")]))
```

Static

```
(define (expect2 type lt rt)
  (cond
    [(and (equal? type lt) (equal? type rt)) type]
    [else (type-case Type type
            [(numT) (error 'tc "needs 2 numbers")]
            [(strT) (error 'tc "needs 2 strings")])]))
```

1. The way `equal?` is used here can only work on `Type` values, not on `Value` values.
2. In a production system, the dynamic checks would likely be removed. The streamlining of the interpreter is part of the point of static typechecking. The typechecker runs once, while the interpreter code potentially runs many times.

Testing the checker

p. 112

tc1

```
(test (tc (binE (plus) (numE 5) (numE 6))) (numT))
(test (tc (binE (++) (strE "hello") (strE "world")))
   (strT))
(test/exn (tc (binE (++) (numE 5) (numE 6))) "strings")
(test/exn (tc (binE (plus) (strE "hello")
                    (strE "world"))) "numbers")
```

Base Cases

p. 113

Corresponding to the base cases of our type checker

```
[(numE n) (numT)]
[(strE b) (strT)]
```

We have the *axioms* for each number `n` and string `s`

- $\vdash \mathtt{n} \colon \mathrm{Num}$
- $\vdash \mathtt{s} \colon \mathrm{Str}$

Base Cases

Corresponding to the base cases of our type checker

```
[(numE n) (numT)]
[(strE b) (strT)]
```

We have the *axioms* for each number n and string s

- $\vdash$ n: Num
- $\vdash$ s: Str

p. 113

1. These are very similar to terminals in a grammar.
2. The $\Gamma \vdash e : T$ is actually a ternary operator. The type environment Γ is actually $\emptyset$ here, but the convention is to omit the symbol in that case.

Conditional rules

p. 114

Sample typechecker case

```
[(plus l r) (if (and (numT? (tc l)) (numT? (tc r)))
                (numT)
                (error 'tc "not both numbers"))]
```

Equivalent type rule

$$\frac{\vdash \texttt{e1} : \mathrm{Num} \qquad \vdash \texttt{e2} : \mathrm{Num}}{\vdash \texttt{(+ e1 e2)} : \mathrm{Num}}$$

Conditional rules

p. 114

Sample typechecker case

```
[(plus l r) (if (and (numT? (tc l)) (numT? (tc r)))
              (numT)
              (error 'tc "not both numbers"))]
```

Equivalent type rule

$$\frac{\vdash \texttt{e1} : \mathrm{Num} \quad \vdash \texttt{e2} : \mathrm{Num}}{\vdash \texttt{(+ e1 e2)} : \mathrm{Num}}$$

p. 114

1. We read this like “if all the things on the top (the *antecedent*) are true, then the thing on the bottom (the *consequent*) is also true”

Type Judgements

p. 116

- Suppose we want to check the type of `(+ 5 (+ 6 7))`.
- We can apply the previous rule, but we are not done

$$\frac{\vdash \texttt{5} : \mathrm{Num} \qquad \vdash \texttt{(+ 6 7)} : \mathrm{Num}}{\vdash \texttt{(+ 5 (+ 6 7))} : \mathrm{Num}}$$

- A second application of the same rule is needed

$$\frac{\vdash \texttt{5} : \mathrm{Num} \qquad \dfrac{\vdash \texttt{6} : \mathrm{Num} \quad \vdash \texttt{7} : \mathrm{Num}}{\vdash \texttt{(+ 6 7)} : \mathrm{Num}}}{\vdash \texttt{(+ 5 (+ 6 7))} : \mathrm{Num}}$$

Type Judgements

p. 116

- Suppose we want to check the type of `(+ 5 (+ 6 7))`.
- We can apply the previous rule, but we are not done

$$\frac{\vdash 5 : \mathrm{Num} \quad \vdash \texttt{(+ 6 7)} : \mathrm{Num}}{\vdash \texttt{(+ 5 (+ 6 7))} : \mathrm{Num}}$$

- A second application of the same rule is needed

$$\frac{\vdash 5 : \mathrm{Num} \quad \dfrac{\vdash 6 : \mathrm{Num} \quad \vdash 7 : \mathrm{Num}}{\vdash \texttt{(+ 6 7)} : \mathrm{Num}}}{\vdash \texttt{(+ 5 (+ 6 7))} : \mathrm{Nu}}$$

p. 116

1. The process starts with the rule for the top level expression
2. Nodes of the proof tree are expanded upwards, until all nodes are *trivial*
3. This expansion corresponds to a trace of a recursive type-checker

(Lack of) Type Judgements

p. 118

Suppose we want to check the type of `(+ 5 (+ 6 "hi"))`.

- We can apply the rule for +, but we are not done

$$\frac{\vdash \texttt{5} : \mathrm{Num} \qquad \vdash \texttt{(+ 6 "hi")} : \mathrm{Num}}{\vdash \texttt{(+ 5 (+ 6 "hi"))} : \mathrm{Num}}$$

- A second application of the same rule is needed

$$\frac{\vdash \texttt{5} : \mathrm{Num} \qquad \dfrac{\vdash \texttt{6} : \mathrm{Num} \quad \boxed{\vdash \texttt{"hi"} : \mathrm{Num}}}{\vdash \texttt{(+ 6 "hi")} : \mathrm{Num}}}{\vdash \texttt{(+ 5 (+ 6 "hi"))} : \mathrm{Num}}$$

- We get stuck because there is no axiom that tells us `"hi"` is a Number.

.

(Lack of) Type Judgements

p. 118

Suppose we want to check the type of `(+ 5 (+ 6 "hi"))`.

- We can apply the rule for +, but we are not done

$$\frac{\vdash 5 : \text{Num} \quad \vdash \texttt{(+ 6 "hi")} : \text{Num}}{\vdash \texttt{(+ 5 (+ 6 "hi"))} : \text{Num}}$$

- A second application of the same rule is needed

$$\frac{\vdash 5 : \text{Num} \quad \dfrac{\vdash 6 : \text{Num} \quad \vdash \texttt{"hi"} : \text{Num}}{\vdash \texttt{(+ 6 "hi")} : \text{Num}}}{\vdash \texttt{(+ 5 (+ 6 "hi"))} : \text{Num}}$$

- We get stuck because there is no axiom
Number.

p. 118

1. Getting stuck is exactly where our type-checker calls `error`
2. The terminology is based on logic, so in this case we fail to reach a judgement
3. So far our proof systems are pretty simple, because we have at most one rule to apply in a given step

Typing conditionals 1/2

p. 119

Suppose we want to typecheck `(if C T E)`. Let's agree

- `C` should be boolean
- `T` and `E` should have the same type
- We need a new construct to express the latter: *type variables*

$$\frac{\vdash \texttt{C} : \mathrm{Bool} \qquad \vdash \texttt{T} : U \qquad \vdash \texttt{E} : U}{\vdash \texttt{(if C T E)} : U}$$

Typing conditionals 1/2 p. 119

Suppose we want to typecheck `(if C T E)`. Let's agree

- `C` should be boolean
- `T` and `E` should have the same type
- We need a new construct to express the latter: *type variables*

$$\frac{\vdash \texttt{C} : \mathrm{Bool} \quad \vdash \texttt{T} : U \quad \vdash \texttt{E} : U}{\vdash (\texttt{if C T E}) : U}$$

p. 119

1. As the book explains, if we don't assume both branches have the same type, things get complicated
2. The assumption that the test is Boolean is more one of strictness (i.e. wanting to catch more type errors) than a simplification

Typing conditionals 2/2

p. 121

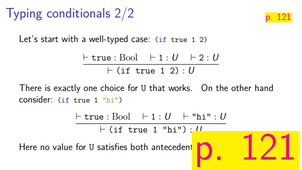
Let's start with a well-typed case: `(if true 1 2)`

$$\frac{\vdash \texttt{true} : \mathrm{Bool} \qquad \vdash \texttt{1} : U \qquad \vdash \texttt{2} : U}{\vdash \texttt{(if true 1 2)} : U}$$

There is exactly one choice for `U` that works. On the other hand consider: `(if true 1 "hi")`

$$\frac{\vdash \texttt{true} : \mathrm{Bool} \qquad \vdash \texttt{1} : U \qquad \vdash \texttt{"hi"} : U}{\vdash \texttt{(if true 1 "hi")} : U}$$

Here no value for `U` satisfies both antecedents



1. The translation of these type variables into code is not as hard as you might think. We just calculate the types of the sub-expressions, then check for equality

Arrow Types

p. 123

$$\frac{\vdash \texttt{F} : (? \rightarrow ?) \qquad \vdash \texttt{A} : ?}{\vdash (\texttt{F}\ \texttt{A}) : ?}$$

$$\frac{\vdash \texttt{F} : (T \rightarrow ?) \qquad \vdash \texttt{A} : T}{\vdash (\texttt{F}\ \texttt{A}) : ?}$$

$$\frac{\vdash \texttt{F} : (T \rightarrow U) \qquad \vdash \texttt{A} : T}{\vdash (\texttt{F}\ \texttt{A}) : U}$$

Function Application Example 1

$$\frac{\vdash \texttt{F} : (T \rightarrow U) \qquad \vdash \texttt{A} : T}{\vdash \texttt{(F A)} : U}$$

$$\frac{\vdash \texttt{(lambda (x : T) 3)} : (T \rightarrow ?) \qquad \vdash \texttt{A} : T}{\vdash \texttt{((lambda (x : T) 3) A)} : ?}$$

- We can't really type check function application with out type rules for function definitions

Type annotations for lambda

lamannote

```
(define f (lambda ([x : Number]) (+ x 1)))
(define g (lambda (x) (+ x 1))) ;; type inference
```

Function definitions

p. 127

$$\frac{\Gamma[V \leftarrow T] \vdash \texttt{body} : ?}{\Gamma \vdash \texttt{(lambda (V : } T\texttt{) body)} : ? \rightarrow ?}$$

Function Application Example 1 redux

$$\frac{\Gamma \vdash \texttt{F} : (T \to U) \qquad \Gamma \vdash \texttt{A} : T}{\Gamma \vdash \texttt{(F A)} : U}$$

$$\frac{\Gamma \vdash \texttt{(lambda (x : T) 3)} : (T \to ?) \qquad \Gamma \vdash \texttt{A} : T}{\Gamma \vdash \texttt{((lambda (x : T) 3) A)} : ?}$$

$$\frac{\dfrac{\Gamma[x \leftarrow T] \vdash 3 : ?}{\Gamma \vdash \texttt{(lambda (x : T) 3)} : (T \to ?)} \qquad \Gamma \vdash \texttt{A} : T}{\Gamma \vdash \texttt{((lambda (x : T) 3) A)} : ?}$$

Function Application Example 1 in plait

apply1

```
(define f (lambda (x) 3))
(define v (f "hello world"))
```