Programming Language Theory

Section 1

21900628 Sechang Jang

1. (3 point) For each chapter, write a one-page summary
2. Answer the following questions in the report
   A. Our FAE implementation does not support type checking now. So the following code is not processed well. {+ 4 {fun {x} x}}
      i. (1 point) Define axioms and rules with antecedents and consequent for FAE that supports two numbers for arithmetic operations (+ and -)
      ii. (1 point) Deploy the judgments as in the textbook page 116 for the case, {+ 4 {fun {x} x}}

1. Introduction to Types

In programming languages, the term "type" refers to the category of a value or expression. Generally, the type in the programming language specifies what kind of data (such as integer or string) can be stored (or mutated).

In the language we are implementing, the type would refer to a static check. Type can be divided into the static type and dynamic type. Static Type refers to “one that can be done purely with the program source”, which means the type can be seen in the code explicitly. In another perspective, in statically typed language, the data type of a variable is known at compile-time. An example would be C or Java. Dynamic type would refer to the opposite concept of static type. Dynamic type is the “one that cannot be done purely with the program source”. In a dynamic type, the data type is determined at runtime. Dynamic conditions and may suffer from type errors in runtime, which we majorly suffered from Python.

The reason why we are adapting the concept of the type is because many languages do not have type or disagree with how the type is defined and used. However, certain aspects generally agree with many programming languages.

Think of types as abstractions of the values your program uses while it's running. At runtime, you might have various numbers, strings, images, and Booleans(true or false). However, when we talk about types, we group these into broader categories, focusing only on the differences between these groups.

Type Checking is growing!

```
(define (tc e)
  (type-case Exp e
    [(binE o l r)
     (type-case BinOp o
       [(plus) (and (tc l) (tc r))])]
    [(numE v) #true]))

(test (tc (binE (plus) (numE 5) (numE 6))) #true)
```

The type-checker currently returns true always because it only deals with numbers and a single operation. To generate errors, there's a need to expand the types and operations. For example, adding a string concatenation operation, ++, type-checking can detect potential issues in programs.

```
(define (tc e)
  (type-case Exp e
    [(binE o l r)
     (type-case BinOp o
       [(plus) (and (tc l) (tc r))]
       [(++)   (and (tc l) (tc r))])]
    [(numE v) #true]
    [(strE v) #true]))

(test (tc (binE (++) (strE "hello") (strE "world"))) #true)
```

The problem it only can determine the specific types of sub-expressions. This limitation becomes evident when dealing with operations like string concatenation (++), where we need to ensure it produces the expected types, strings.

```
(define (tc e)
  (type-case Exp e
    [(binE o l r)
     (type-case BinOp o
       [(plus) (if (and (numT? (tc l)) (numT? (tc r)))
                   (numT)
                   (error 'tc "not both numbers"))]
       [(++)   (if (and (strT? (tc l)) (strT? (tc r)))
                   (strT)
                   (error 'tc "not both strings"))])]
    [(numE v) (numT)]
    [(strE v) (strT)]))
```

(tc : (Exp -> Type)),

"Type is a new (plait type) definition that records the possible types" : A type-checker works with "weak" values, as demonstrated by the numE case which disregards the specific numeric values.

The transition from a type-checker to a type-calculator involved transforming the Boolean output into actual types for each expression, replacing #true with a Type and #false with an error in mathematical terms.

A Concise Notation

We'll express terms in the format **|- e : T**, where "e" represents expressions, "T" stands for types, and ":" is read as "has type."

```
|- e1 : Num     |- e2 : Num
-------------------------
|- (+ e1 e2) : Num
```

The expression |- e1 : Num and |- e2 : Num implies that if e1 has type Num and e2 has type Num, then the expression (+ e1 e2) has type Num.

" The part above is called the antecedent (that which goes before) and the part below is called the consequent (that which comes after). "

2. Growing Types: Division, Conditionals

Division (except addition, multiplication, and subtraction which consumes two numbers) is a partial function. We need to consider the case when the denominator is zero. To address this issue, we could use several strategies.

1. We can declare its return type as (Option of Number), requiring explicit checks.

2. We can restrict division to non-zero second arguments, a significant change to the type system that demands callers prove non-zero inputs.

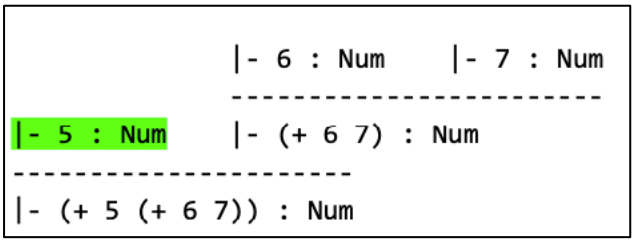
3. Assigning division the same type as other binary numeric operations and addressing exceptional cases with errors or exceptions, putting the burden on the program.

Most programming languages choose the third strategy, but what if we considered the first and second cases? In the second case, languages attempt to prove non-zero-ness and, when unsuccessful, shift the burden to the programmer.

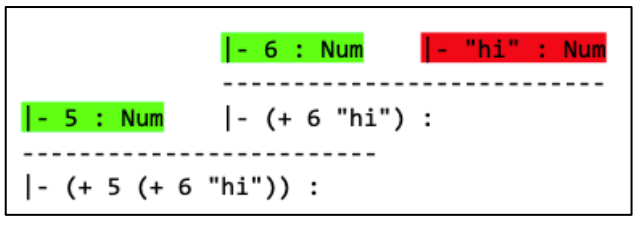
From the outside perspective of seeing the type as the abstraction of the values, we might see the type as a "static discipline".

From Axioms and Rules to Judgments

The previous rule that we discussed does not follow any axioms. "Program does not match the syntax of a single number or string".
If we are using the conditional rule, we could see the following procedure.



The program is considered checked type successfully, when every part of the tree terminates in an axiom, forming a judgment. "Judgement" is similar to the type-checker's execution pattern, confirming it as type-checked and producing a Num value. The difference is we are able to skip the details of "passing and returning", using pattern-matching.



Consider the "if ... and ... then" interpretation. If we cannot fulfill all the antecedents, we are unable to establish anything about the consequents, resulting in an incomplete tree.

Therefore, a type error is a simple failure to construct a judgment. A judgment is only assigned when the tree is fully "checked off," meaning all antecedents are generated using provided rules, and the leaves consist of actual axioms.

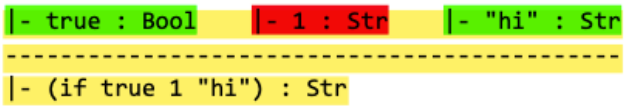
To enhance the type-checking process, we're introducing a rule for the "if" statement. Many languages require the conditional clause to be a Boolean to catch type errors effectively. The goal is to add a type rule, so we need a condition rule. We need the antecedent of sub-expression.

To do so, there are two common solutions.

1. Introduce a type that represents a choice, which is straightforward but adds complexity for code handling such values.

2. Just make a rule to both branches have the same type.

The following is required, and the computation worked like this.
Compute the type of T and E, make sure T and E have the same type, and make this (same) type the result of the conditional.
Then, according to the textbook, it would be like "if C has type Bool and T has type U and E has [the same] type U, then (if C T E) has [the same] type U".



"there is no way to construct a judgment for this program, it too has a type error.

Where Types Diverge from Evaluation

In the evaluation process, a conditional statement examines and executes only one branch since it conditionally executes code. In contrast, a type-checker traverses both branches, highlighting a distinction in traversal strategies between an evaluator and a type-checker.

1. (1 point) Define axioms and rules with antecedents and consequent for FAE that supports two numbers for arithmetic operations (+ and -)

   A. Axioms

      i. |- e : T

         1. Means e has type T

         2. |- : anything at all

      ii. |- n : num

   B. Rules

      i. |- e1: num |- e2: num

      |- {+ e1 e2}: num

      ii. |- e1: num |- e2: num

      |- {- e1 e2): num

2. (1 point) Deploy the judgments as in the textbook page 116 for the case, {+ 4 {fun {x} x}}

|- 4: num |- {fun {x} x}}: num

|- {+ 4 {fun {x} x}} :