CS 536 Announcements for Wednesday, March 20, 2024

Midterm 2
• Thursday, March 21, 7:30 – 9 pm
• S429 Chemistry
• bring your student ID

Last Time
• name analysis
  • handling tuples
  • handling classes
• review for Midterm 2

Today
• type checking
• type-system concepts
• type-system vocabulary
• base
  • type rules
  • how to apply type rules

After Spring Break
• runtime environments

What is a type?
Short for data type
• classification identifying kinds of data
• a set of possible values that a variable can possess
• operations that can be done on member values
• a representation (perhaps in memory)

Type intuition – is the following allowed?
int a = 0;
int *pointer = &a;
float fraction = 1.2;
a = pointer + fraction;
Components of a type system

base types (built-in/primitive)  
integer logical void

rules for constructing types  
tuple struct class enum

means of determining if types are compatible or equivalent

Can values with different types be combined? If so, how?
  eg. 7 + 1.2

Can we consider two types the same? If so, how?
  tuple Point &
  integer x, y
  tuple Pair &
  integer a, b

rules for inferring the type of an expression

Type rules of a language specify

What types the operands of an operator must be (+, >, ==, =)

double a;
int b;
a = b; ← allowed in Java, C++
b = a; ← not legal in Java, legal in C++

What type the result of an operator is

Type coercion
  • implicit cast from one data type to another
    int j = 3.0; ← may result in information loss
  • type promotion – destination type can represent source type
    double f = 123;

Places where certain types are expected

if (x = 4) {
  ...  
}

In Java, condition of if must evaluate to logical
Type checking: **when do we check?**

**static typing** – type checking done at **compile time**

**dynamic typing** – type checking done at **runtime**

**combination of the two** ← Java does this

```
Fruit
  ↑
Apple      Orange

Apple a = new Apple();
Fruit f = a;
```

- upcasting

```
Apple a = new Apple();
Orange o = (Orange)a;
```

- cross-casting - not allowed - static check

```
Fruit f = new Apple();
if (...) {
  f = new Orange();
}
```

- down-casting - compiles - runtime error if f is not an Orange

```
Apple two = (Apple)f;
```

**Static vs dynamic trade-offs**

- **static**
  + compile-time error checking
  + compile-time optimizations may be more available

- **dynamic**
  + avoid dealing with errors that don’t matter
  + some added flexibility
  - failures can happen at runtime

**Duck typing** - type is defined by methods and properties - often done dynamically

```python
class bird:
    def quack() : print("quack")

class robobird
    def quack() : print("0100101101")
```
Type checking: what do we check?

strong vs weak typing — continuum with no precise defs
- degree to which type checks are performed
- degree to which type errors are allowed to happened at runtime

General principles
- statically typed \( \rightarrow \) stronger (fewer type errors possible at runtime)
- more implicit casting allowed \( \rightarrow \) weaker
- fewer checks performed at runtime \( \rightarrow \) weaker

Example

C (weaker)

```c
union either {
    int i;
    float f;
} u;

u.i = 12;

float val = u.f;
```

Standard ML (stronger)

```ml
real(2) + 2.0
```

Type safety

- All successful operations must be allowed by the type system
- Java is explicitly designed to be type safe — if you have a variable declared to be of some type, then it is guaranteed (at runtime) to be of that type (or a sub-type of that type)
- C is not
  - it allows \( \rightarrow \) (type-safety violations)
    ```c
    printf("%s", 1);
    struct big {
        int a[100000];
    };
    struct big *b = malloc(1);
    ```
  - memory safety issue
- C++ is a little better
  ```c
  class T1 { char a; }
  class T2 { int b; }
  int main() {
      T1 *myT1 = new T1();
      T2 *myT2 = new T2();
      myT1 = (T1 *)myT2;  \( \rightarrow \) allows unchecked casts
  }
  ```
Type checking in base

**base's type system**
- primitive types
  - integer, logical, void, string
- type constructors
  - tuple
- coercion
  - logical cannot be used as an integer & vice versa

**Type errors in base**

Operators applied to operands of wrong type
- arithmetic operators **must have integer operands**
- logical operators **must have logical operands**
- equality operators `==` `!=`
  - must have operands of the same type
  - can't be applied to
    - function names
    - tuple names
    - tuple variables
- other relational operators
- assignment operator `=`
  - must have integer operands
  - can't be applied to
    - function names
    - tuple names
    - tuple variables

Expressions that, because of context, must be a particular type but are not
- expressions that must be logical (in base)
  - condition of if, condition of while
- reading
  - `read ?? x` **x can't be function name, tuple name, tuple variable**
- writing
  - `write <<< x` **x can't be function name, tuple name, tuple variable**
  - but can be string, expression eg `(7+3)`

Related to function calls
- invoking (i.e., calling) something that is not a function
- invoking a function with
  - wrong number of arguments
  - wrong types of arguments
- returning a value from a **void function**
  - ie, can't have return `x`
- not returning a value from a **non-void function**
  - ie, can't have return
- returning wrong type of value in a **non-void function**
Type checking
Recursively walks the AST to
- determine the type of each expression and sub-expression using the type rules of the language
- find type errors (report them)
Add a `typeCheck` method to AST nodes

Type checking: binary operator

Type "checking": literal

Type checking: `IdNode`

Type checking: others
- call to function `f`
  - get type of each actual parameter of `f`
  - match against type of corresponding formal parameter of `f`
  - pass `f`'s return type up the tree
- statement `s`
  - type check constituents of `s`
Type checking (cont.)

Type checking: errors

Goals:
- report as many distinct errors as possible
- don't report same error multiple times – avoid error cascading

Introduce internal error type
- when type incompatibility is discovered
  - report the error
  - pass error up the tree
- when a type check gets error as an operand
  - don't (re)report an error
  - pass error up the tree

Example:

```plaintext
integer a.
logical b.
a = True + 1 + 2 + b.
b = 2.
```