CS 536 Announcements for Thursday, April 7, 2022

Last Time
- name analysis
- static vs dynamic scoping
- scoping issues to consider
- name analysis in minim
  - scoping rules
  - symbol table
  - handling structs

Today
- type checking
- type-system concepts
- type-system vocabulary
- minim
  - type rules
  - how to apply type rules

Next Time
- runtime environments

Name analysis and structs

Defining a struct

Declaration of a variable of a struct type

Accessing fields of a struct
- Compiler needs to
  - for each field: determine type, size, and offset with the structure
  - determine overall size of structure
  - verify declarations and uses of something of a struct type are valid
- Idea: each struct type definition contains its own symbol table for its field declarations
  - associated with the main symbol table entry for that struct's name

![Diagram of a tree with nodes labeled a, b, c, and a numbered structure diagram with steps 1 and 2 for accessing nodes, and notes on process access and child handling.]

1. If L child is id:
   - check id, get sym tab for struct, lookup R child
2. If L child is access:
   - process L child
   - if sym tab is not null, lookup R child

If R child is struct, set ref in access node to struct's sym tab
else set to null
Handling Classes

Similar to handling aggregate data structures
- also need to be able to search the class hierarchy
to see if uses are to inherited fields & methods

Idea:
Symbol table for each class with two nesting hierarchies
1) for lexical scoping within methods (i.e., "regular" sym tab)
2) for inheritance hierarchy
   - not just a list of hash-tables - hierarchy not necessarily linear

To resolve a name
   - first look in lexical scoping sym tab (i.e., "regular" one)
   - then search inheritance hierarchy

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?

```c
int a = 0;
int *pointer = &a;
float fraction = 1.2;
a = pointer + fraction;
```
Components of a type system

base types (built-in/primitive)

- int, bool, void

rules for constructing types

- structs, classes

means of determining if types are compatible or equivalent

- Can values with different types be combined? If so, how?
- Can we consider 2 types the same? If so, how?

   - struct point
     - int x;
     - int y;
   - struct pair
     - int a;
     - int b;

rules for inferring the type of an expression

Type rules of a language specify

- what types the operands of an operator must be

  Example:
  
  ```java
  double a;
  int b;
  a = b;  // legal 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
    ```java
    int j = 3.0;
    ```
    - may have information loss
  - type promotion
    - destination type can represent source type
    ```java
    double f = 123;
    ```

- places where certain types are expected

  ```java
  if (x == y) 
  ```
  
  ok as minima expression
  - allows x = y = z = 8
  
  But in minimum cond of if must be boolean

  ```java
  if (true) 
  ```
  
  - may have information loss
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
  f = new Orange();

^ Apple two = (Apple)f;

downcasting - compiles runtime error if f is an Orange
```

Static vs dynamic trade-offs

- static
  + compile-time error checking
  + compile-time optimization

- dynamic
  + avoid dealing with errors that don't matter
  + some add flexibility
  - failures can happen at runtime

Duck typing - type is defined by methods & properties

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

class robobird:
  def quack(): print("0110101101")
```

often dynamically checked
Type checking: what do we check?

strong vs weak typing
- degree to which type checks are performed
- degree to which type errors are allowed to happened at runtime

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

Example

\[
\begin{align*}
C & \quad \text{(weaker)} \\
\text{union either } & \quad \text{either } \\
\text{int } & \quad \text{int } \\
\text{float } & \quad \text{float } \\
\text{3u.j} & \quad \text{3u.j} \\
\text{u.i = 12.j} & \quad \text{u.i = 12.j} \\
\text{float val = u.f.j} & \quad \text{float val = u.f.j} \\
\end{align*}
\]

StandardML (stronger)
\[
\begin{align*}
\text{real(2) + 2.0} \\
\end{align*}
\]

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 with some type, then it is guaranteed to be of that type.
- C is not

allows
(type-safety violators)

\[
\begin{align*}
\text{printf} \left( \text{"%s"}, 1 \right) \\
\text{struct big } \& \\
\text{int a[100000]j} \\
\text{3j} \\
\text{strum big * b = malloc(1)j} \\
\end{align*}
\]

memory safety

C++ is a little better

allows unchecked casts

\[
\begin{align*}
\text{class T1 &char a;} & \;
\text{class T2 &int b;} \\
\text{int main } & \;
\text{int main } \\
\text{T1 * myT1 = new T1();} & \;
\text{T2 * myT2 = new T2();} \\
\text{myT1 = (T1 *)myT1;} & \;
\text{myT2 = (T1 *)myT2;} \\
\end{align*}
\]
Type system of minim

**minim's type system**
- primitive types: `int, bool, void, string`
- type constructors: `struct`
- coercion: `bool cannot be used as an int & vice-versa`

Type errors in minim

**Operators applied to operands of wrong type**
- arithmetic operators: `must have int` operands
- logical operators: `must have bool` operands
- equality operators: `==` `!=`
  - `must have operands of same type`
  - `can't be applied to`
    - function names (but can be applied to function results)
    - struct names
    - struct variables
- other relational operators: `must have int` operands

**Assignment operator**: `=`
- `must have operands of same type`
- `can't be applied to`
  - function names
  - struct names
  - struct variables

**Expressions that, because of context, must be a particular type but are not**
- expressions that must be boolean (in minim)
  - condition of if, condition of while
- reading: `input >> x;`
  - `x` can't be function, struct name, struct variable
- writing: `disp << x;`
  - `x` can't be function, struct name, struct variable
  - but can be an expression e.g. `(7*3)`
Type errors in minim (cont.)

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
- not returning a value from a non-void function
- 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

Add a `typeCheck` method to AST nodes

Type checking: binary operator

Add a `typeCheck` method to AST nodes

Type "checking": literal

- Cannot be wrong
  - just pass type of literal up the tree

Type checking: `IdNode`

- Look up type of declaration
  - should be a symbol linked to node
- Pass symbol type up the tree
Type checking (cont.)

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 $s$

  - nothing to pass up tree — a statement does not produce a value — $s$ has no "return type"

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:
```c
int a;
bool b;
a = true + 1 + 2 + b;
b = 2;
```