CSE401: Semantic Analysis (D)

David Notkin
Autumn 2000

Today

- Miscellaneous issues in type checking
- A status check: where are we, and where are we going?

Type checking

- We've covered the basic issues in how to check semantic, type-oriented, properties for the data types and constructs in PL/0 (and some more)
- But there are other features in languages richer than PL/0, and we'll looking at some of them today

Records

- Records (or structs) combine heterogeneous types into a single (usually named) unit
- ```
  record R = begin
    x : int;
    a : array[10] of bool;
    m:char;
  end record;

  var t : R;

  …  r.x
  ```


Type checking records

- Need to represent record type, including fields of record
- Need to name user-defined record types
- Need to access fields of record values
- May need to handle unambiguous but not fully qualified names (depending on language definition)

An implementation

- Representing record type using a symbol table for fields
  - `class RecordType: public Type {..};`
  - Create `RecordTypeSTE`
- To typecheck `expr.x`
  - Typecheck `expr`
    - Error if not record type
  - Lookup `x` in record type's symbol table
    - Error if not found
  - Extract and return type of `x`
  ```

Type checking classes or modules

- A class/module is just like a record, except that it can contain procedures in addition to simple variables
- So they are already supported by using a symbol table to store record/class/module fields
- Procedures in the class/module can access other fields of the class/module
  - But this is already support by nesting procedures in record symbol table

Type equivalence

- When is one type equal to another?
  - Implemented in PL/0 with `Type::same` function
- It's generally "obvious" for atomic types like `int`, `string`, user-defined types
- What about type constructors like arrays?

```
var a1    : array[10]  of int;
var a2,a3 : array[10]  of int;
var a4    : array[20]  of int;
var a5    : array[10]  of bool
var a6    : array[0:9] of int;
```


Structural equivalence

- Two types are equal if they have the same structure
  - If atomic types, then obviuos
  - If type constructors
    - Same constructor
    - Recursively, equivalent arguments to constructor
- Implement with recursive implementation of `same`

Name equivalence

- Two types are equal if they came from the same textual occurrence of type constructor
- Implement with pointer equality of `Type` instances
- Special case: type synonyms don't define new types

`same` & `different`

```
class Type {
public:
  …
  virtual bool same(Type* t) = 0;
  bool different(Type* t) { return !same(t); }
  …
};
class IntegerType : public Type {
public:
    …
    bool same(Type* t) { return t->isInteger(); }
    …
};
```


Implement structural equivalence

details

- Problem: want to dispatch on two arguments, not just receiver
  - That is, choose what method to execute based on more than the class of the receiver
- Why? There's a symmetry that the OO dispatch approach skews
  - `if (lhs->different(rhs)) {…error…}`
- Why not: `if (different(lhs,rhs)) {…error…}`

Multi-methods

- Languages that support dispatching on more than one argument provide *multi-methods*
- For example, they might look like
  - `virtual bool same(type* t1, type* t2) {return false;}`
  - `virtual bool same(IntType* t1, IntType* t2) {return true;}`
  - `virtual bool same(ProcType* t1, ProcType* t2) {return same(t1->args,t2->args);}`
- Different from static overloading in C++

Overloading: quick reminder

- Overloading arises when the same operator or function is used to represent distinct operations
  - `3 + 4`
  - `3.14159 + 2.71828`
  - `"mork" + "mindy"`
- The compiler statically decides which "+" to compile to based on the (type) context

Polymorphism: quick reminder

- Polymorphism is different from overloading
- In overloading the same operator means different things in different contexts
- In polymorphism, the same operator works on different types of data
  - `(length '(a b c)) vs. (length '((a) (b c) 3 4))`
  - `(sort '(4 1 2)) vs. (sort '(c g a))`
- In polymorphism, the compiler compiles the same code regardless

But C++ has no multi-methods:

So we can use double dispatching

```
class Type {
  virtual bool same(Type* t2) = 0;
  virtual bool sameAsInteger(IntegerType* t1) {
    return false;}
  virtual bool sameAsProc(ProcType* t1) {
    return false;}
};
class IntegerType : public Type {
  bool same(Type* t2) {
    return t2->sameAsInteger(this);}
  bool sameAsInteger(IntegerType* t1) {
    return true;}
};
```


Type conversions and coercions

- In C, can explicitly convert data of type `float` to data of type `int` (and some other examples)
  - Represent it explicitly as a unary operator
  - Type checking and code generation work as normal
- In C, can also implicitly coerce
  - System must insert unary conversion operators as part of type checking
  - Code generation works as normal

Type casts

- In C and Java (and some other languages) can explicitly cast an object of one type to another
  - Sometimes a cast means a conversion (e.g., casts between numeric types)
  - Sometimes a cast means just a change of static type without any computation (e.g., casts between point types)

Safety of casting

- In C, the safety of casts is not checked
  - That is, it's possible to convert into a representation that is illegal for the new type of data
  - Allows writing of low-level code that's type-unsafe
  - More often used to work around limitations in C's static type system
- In Java, downcasts from superclass to subclass include a run-time type check to preserve type safety
  - This is the primary place where Java uses dynamic type checking

Where are we?

- We now know, in principle, how to
  1. take a string of characters
  2. convert it into an AST with associated symbol table
  3. and know that it represents a legal source program (including semantic checks)
- That is the complete set of responsibilities (at a high-level) of the front-end of a compiler

Normally…

- …we'd now take a break for a mid-term exam
- But because of my travel schedule, we'll delay the mid-term for two weeks
- Arguably, this is better because you'll have more implementation experience with the front-end by then
- Arguably, this is worse because you'll forget what was in the lectures and the book
- Unarguably, the mid-term will be Wednesday November 8^{th}, with a review on Monday November 6^{th}
  - You'll be voting on the 7^{th}, too

Next…

- …what to do now that we have this wonderful AST representation
- We'll look mostly at interpreting it or compiling it
  - But you could also analyze it for program properties
  - Or you could "unparse" it to display aspects of the program on the screen for users
  - …