CIS 341: Compilers

Lecture 20

The Plan

- Today:
  - Modularity, Encapsulation, Objects

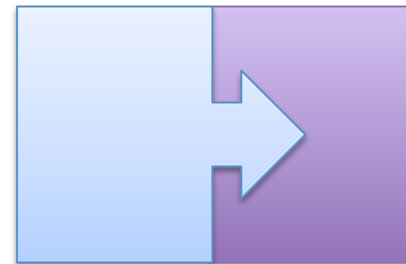
- Project 4: Code generation & typechecking for the "AT" (arrays and types) language
  - Due: Friday October 24th at 11:59 pm

Course Summary So Far

- We've seen how to compile simple languages:
  - Functions, procedures, first-class functions
  - Arrays, records, data types
  - Simple type checking, code generation
- Next:
  - User-defined abstract data types: modules, objects
- Then:
  - Optimizations: register allocation, dataflow analysis, etc.
- Time permitting:
  - Exceptions, Garbage Collection, Run-time systems, Parametric Polymorphism, Type Inference

Modular Programming

- Programs are typically composed of many modules.
  - Separate compilation – scalable to millions of lines
  - Code reuse – libraries, sharing
  - Namespace management
  - Encapsulation – hiding complexity
  - Abstraction & abstract data-types
  - Security



- What is a module?
  - A collection of named, related values and types
  - Definitions (partially) hidden from the outside
- Examples: Java classes & packages, C++ classes, Modula-3 modules, SML/Ocaml structures & functors, CLU clusters, C source files, …

Separate Compilation

- Program is made of several *compilation units*
  - Independent inputs to the compiler

- Avoids needing to recompile the whole program for every change
- Code is more reusable (libraries)
- Examples:
  - C: .c files / Java: .java files / OCaml: .ml files

- For building a whole program out of compilation units:
- Need to know how to reference values in other units
  - Solution: namespaces + linking
- Need to know datatype sizes (for code generation) or types (for type safety)
  - Solution: interfaces (C: .h files / Java: .class files / OCaml: .mli files)

Namespaces

- In C and FORTRAN: all global identifiers are visible everywhere
- Problem:
  - Can't have two global variables or functions with the same name
  - (Also, linker doesn't type check)
- Solutions:
  - C++, Java qualified identifiers: C.x or $P_1.P_2.P_3$.C.x (where C is a class name)
  - Modula-3, OCaml: qualified identifiers + renaming
  - Java, Modula-3, OCaml: link-time type checking

- Wrinkle: object code formats typically have a flat name space
  - Need to *mangle* qualified identifiers
  - e.g. C++: **`int C::f(int x)`** becomes **`f__1Ci`**

Linking

Input: File f1.c File f2.c

```
extern int x;

void main() {
  printf("%d", x);
}
```

```
int x = 341;
```

compiles to asm: f1.s f2.s

assembles to obj: f1.o f2.o

linker a.out

- *Problem:* compiler can't generate code to access variable x because its address is unknown.
- *Solution:* Generate placehold reference to x in f1.s, generate definition of x in f2.s, linker patches the files together, replacing placeholders in f1.s with actual value from f2.s
  - Exact mechanism depends on linker/OS object file format

Encapsulation

- It's often useful to hide some information contained in a module.
- Example:

```
String[] names;       // should be hidden
String[] passwords;   // should be hidden
bool check_password(String n, String p) {
  int j = 0;
  while (j < names.length) {
    if (names[j] == n & passwords[j] == p)
      return true;
    j = j + 1;
  }
  return false;
}
```

- Encapsulation can protect a module's data from tampering
  - Good software engineering practices rely on encapsulation.

Encapsulation Mechanisms

- Fundamentally, need a way to indicate which identifiers should be exported from a module.
- C++/Java: "public" vs. "private" qualifiers:

```
class PWChecker {
private String[] names;       // should be hidden
private String[] passwords;   // should be hidden
public bool check_password(String n, String p) {…}
```

- ML / Modula-3: separate interface (omits hidden identifiers):

```
module type PWChecker = sig
  val check_password : String * String -> bool
  (* Note: no declaration for names or password *)
end
```

- C: "static" qualifier

```
  static int check_password(char *n, char *p)
```

Modules as Records

- Records (or structs) bundle values together, mapping names to values.
- Modules *also* bundle values together…
  - Except that modules are computed a *load* time
  - They are (usually) 2nd class (e.g. modules cannot be passed arguments to functions)
- But… module interfaces look like record types:

```
module PWC = struct
  let names : string array = …
  let passwords : string array = …
  let check_password (n:string, p:string):bool = …
  let is_name (n:string):bool = …
end :
sig
  val check_password : string * string -> bool
  val is_name : string -> bool
end
```

More on Encapsulation

- Example: sets of integers
  - operations: empty, insert, has

In OCaml:

```
type intset = int list
let empty = []
let insert i s = i::s
let rec has i s =
  match s with
  | [] -> false
  | (j::rest) -> if i == j then true else has i rest
```

- Problem: can’t write down the interface unless
  - We expose the implementation of intset as equal to int list
  - Or, alternatively, we expose intset as an *abstract* type

Alternate Implementation of Integer Sets

- Consider this alternate implementation of integer sets as binary search trees:

```
type intset = Leaf | Node of intset * int * intset
let empty = Leaf
let rec insert i s =
  match s with
  | Leaf -> Node(Leaf, i, Leaf)
  | Node(left, j, right) ->
     if i = j then s else
     if i < j then Node(insert i left, j, right)
       else Node(left, j, insert i right)
let rec has i s =
  match s with
  | Leaf -> false
  | Node(left, j, right) ->
     if i = j then true else
       if i < j then has i left
       else has i right
```

Problem of Exposed Representations

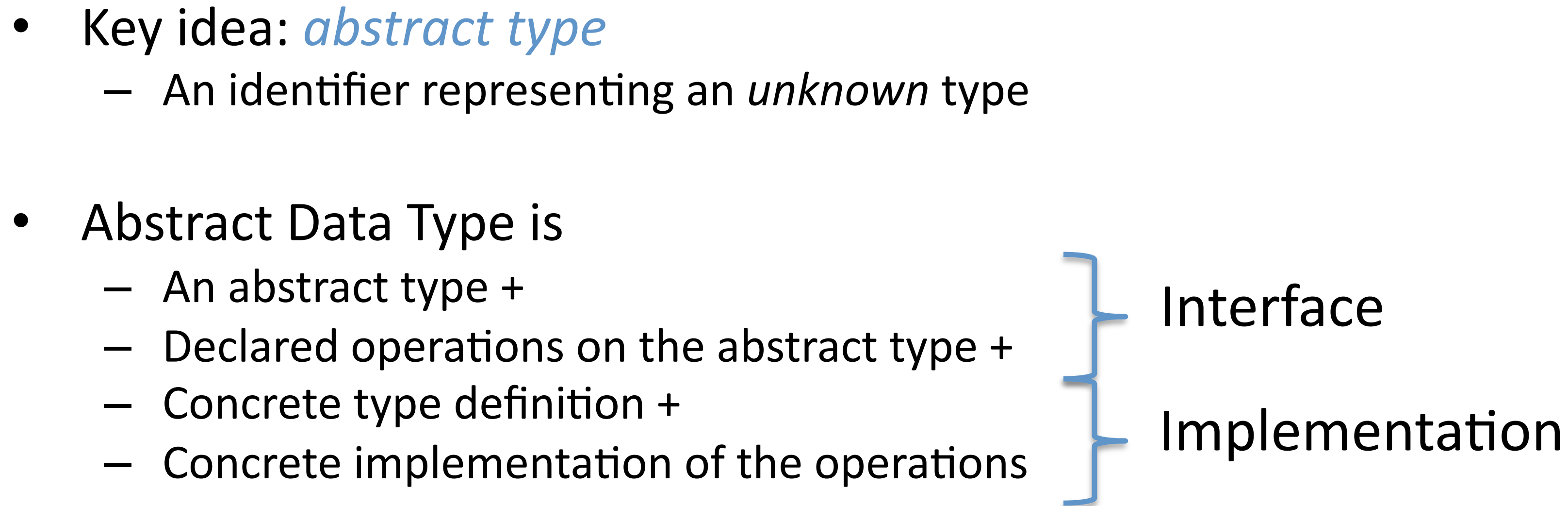
- If we expose the representation type:
  `intset = Leaf | Node of intset * int * intset`
- Client code can break the representation invariant that **`intset`** is a search tree.
  - Concretely, a client could construct a value of type **`intset`** such as:
    `let bad = Node(Leaf, 10, Node(Leaf, 5, Leaf))`
  - Note that "**`has 5 bad`**" will return **`false`**, even though 5 appears as a node in the tree.

- We need encapsulation of values *exported* from the module, not just components inside the module.
  - Only way to create **`inset`**s is via the operations in the interface.

Abstract Data Types

- Key idea: *abstract type*
  - An identifier representing an *unknown* type

- Abstract Data Type is
  - An abstract type +
  - Declared operations on the abstract type +
  - Concrete type definition +
  - Concrete implementation of the operations



- IntSet interface in OCaml:

```
module type IntSet = sig
  type intset         (* Note: no type definition *)
  val empty : intset
  val insert : int -> intset -> intset
  val has : int -> intset -> bool
end
```

IntSet example in OCaml

```
module IntSet1 : IntSet = struct
  type intset = int list
  let empty = []
  let insert i s = i::s
  let rec has = …
end

module IntSet2 : IntSet = struct
  type intset = Leaf | Node of intset * int * intset
  let empty = Leaf
  let rec insert i s = …
  let rec has = …
end
```

This signature ascription *seals* the modules with an abstract type, hiding the representation of intset.

Implementing Abstract Types

- Representation of the abstract type is hidden from code other than the implementation itself
  - CLU, Ada, Modula-3, ML
- Because external code doesn't know representation, it can't violate the abstraction boundary
  - e.g. break representation invariants
- Positive: The same interface can be reimplemented multiple ways.
- Negative: Compiler doesn't know representation either
  - When compiling external code it must use level of indirection
  - No stack allocation of abstract types

IntSet Example in Java

```
public interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
}

class IntSet1 implements IntSet {
    private List<Integer> rep;          // note hidden state

    public IntSet1() {
      this.rep = new LinkedList<Integer>();
    }

    public IntSet1 insert(int i) {
      rep.add(new Integer(i));
      return this;
    }

    public boolean has(int i) {
      return rep.contains(new Integer(i));
    }

}
```

Classes in C++/Java

- Classes have private/public visibility qualifiers that hid part of the object.
- A class is a *partially* abstract type
  - (Note: do not confuse with Java's 'abstract' keyword)
- Interface file declares the representation
  - Method code is (mostly) hidden from the outside
- Positive: This mechanism allows external code to know how much space each object takes while still providing encapsulation
  - Objects can be stack allocated (good for cache coherence/ performance)
- Negative: Change to representation can require complete recompilation, even of external code

IntSet example in C

- intset.h:

```
struct intset;
extern struct intset *empty;
struct intset *insert(int i, struct intset *s);
int has(int i, struct intset *s);
```

- intset.c:

```
#include "intset.h"

struct intset {struct intset *left;  int val; struct
   intset *right; };

struct intset *empty = NULL;

struct intset *insert(int i, struct intset *s) {…}
int has(int I, struct intset *s) {…}
```

No Abstraction in C

- C provides hiding/encapsulation but no abstraction.
- (Unchecked) Casts allow any client code to violate the representation invariants of the module.